[redacted]

10.84
l6N
.743
cop 2

Math

Report No. UIUCDCS-R-75-743

ANALYZING SMOOTH FLOWCHARTS: TEACHING STRUCTURED PROGRAMMING IN A COMPUTER-BASED EDUCATION ENVIRONMENT

by

Daniel Clair Hyde

June, 1975

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-75-743

ANALYZING SMOOTH FLOWCHARTS: TEACHING STRUCTURED PROGRAMMING IN A COMPUTER-BASED EDUCATION ENVIRONMENT*

by

Daniel Clair Hyde

June, 1975

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, June, 1975.

To my wife Mary Jane

because without her encouragement

this thesis would not be possible.

ACKNOWLEDGEMENT

The author wishes to express his gratitude to Professor Sylvian R. Ray for his guidance during the preparation of this thesis.

I would like to thank the Department of Computer Science and the Computer-based Education Research Laboratory for their financial support.

Thanks are also due to my colleagues Prabhaker Mateti and Kishor Trivedi for valuable discussions during the course of this research.

Finally, I wish to thank my wife Mary Jane for her encouragement and patience. I am also indebted to her for typing the several drafts of this thesis.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

Chapter One

INTRODUCTION

1.1 Statement of Research Problem

Many students in introductory programming courses are frustrated by homework assignments run on the computer. It has been observed that a major source of this frustration is the student's inability to plan the flow of ideas, i.e., develop the algorithm. For example, many students if given a flowchart can write the programming code, but find it hard to derive the flowchart. Recent progress in software engineering [Dahl, Dijkstra, and Hoare, 1972] has formulated structured programming, an approach to developing algorithms. Incorporating flowcharts into the structured programming paradigm proves to be a viable solution for beginning programmers, especially nonmajors, in an introductory computer science course. Although structured programming supplies a systematic approach, it is not enough to alleviate the student's frustration. The student requires individual attention at each step to see where he/she erred. One fact is clear: human graders are too scarce a resource for the required individual attention. With the new method proposed in this thesis, this individual attention is now possible for grading student-generated flowcharts in a computer-based education environment.

Any scheme that involves humans grading flowcharts in the traditional large lecture format will be costly in man-hours. Ideally, a machine should handle the drudgery of grading the many flowcharts from hundreds of students. What general characteristics of a machine are needed? It must, at least, have the capability both to allow students to input flowcharts quickly and easily, and to grade the flowcharts, giving fast feedback on errors. This implies a large interactive system with a graphic terminal that has a graphic input device such as a light pen. The PLATO IV system at the University of Illinois has the above characterisitics.

1.2 The PLATO IV Computer-Based Educational System

PLATO IV [Alpert and Bitzer, 1970][1] is a computer-based education system using a dedicated CDC Cyber 73 having 1008 terminal ports. The terminal, designed at the University of Illinois [Stifle, 1970], uses a plasma panel 8.5 inches by 8.5 inches which has full graphics capability accomplished by selectively writing or erasing one of 262,144 dots (512 x 512). An optional feature of the terminal is a touch panel mounted over the screen. This touch panel uses a 16 x 16 grid of infrared lights which allows one to point with his/her finger at 256 distinct areas on the screen. This input device greatly improves the person-machine interface while the student is drawing flowcharts.

[1]References in the text are denoted by author(s)'s last name(s) and date of article.

1.3 The Constraints in a Computer-Based Education Environment

A time sharing system having hundreds of terminals[2] to service must have constraints on processing. Further constraints in storage and processing arise because of the computer-based education (also referred to as computer assisted instruction (CAI)) environment. Numerically, these constraints are 5000 machine instructions per second for processing, 8000 words (60 bit/word) for program, and 150 words per student of addressable storage for data. Although for traditional computer-based education materials, e.g., tutorials, these constraints are adequate, they are totally unsuitable for most approaches used in artificial intelligence programs. A further limitation of the computer-based education environment is that all programming must be done in the authoring language TUTOR. TUTOR is a good authoring language, but because it is not designed to be a general purpose scientific language, it lacks such features as pointers, strings, and local variables. Any list processing, string manipulation, or stack manipulation must be programmed by the programmer at a primitive level similar to programming in FORTRAN. In spite of these constraints, the PLATO IV system is a viable system with hundreds of full graphics interactive terminals available on which to teach large classes of

[2]PLATO IV currently has over 900 terminals connected, of which 250 are on the Urbana-Champaign campus.

introductory-level programmers. Utilizing this large resource, the Department of Computer Science at the University of Illinois is implementing major portions of their CS100-level courses on PLATO [Nievergelt, Reingold, and Wilcox, 1973].

1.4 Teaching via Structured Programming

Since a computer-based education system is being used to grade student flowcharts, it is natural to ask: Can it instruct as well as grade? The answer is in the affirmative. To teach students via machine how to plan the flow of their programs, i.e., generate flowcharts, one must have a systematic approach. The teaching technique used in this research is tailored after the work of Dijkstra [Dahl, Dijkstra, and Hoare, 1972] and others. The author's version of the step-wise refinement technique of structured programming begins with a statement of the problem, breaks it into subproblems, and continues until the problem is solved. To facilitate the technique of step-wise refinement, Dijkstra proposed using a limited control structure--DOWHILE, IFTHENELSE, and assignment statements only. This thesis imposes Dijkstra's limited control structures on flowcharts. This subset of flowcharts, called "smooth flowcharts," is essentially D-flowcharts (D stands for Dijkstra) of the literature [Mills, 1975]. A detailed description of smooth flowcharts will appear in Chapter Two. The instructional portion of the computer-based education program attempts, in about two hours of student contact time, to teach the systematic approach of step-wise refinement in the framework of smooth flowcharts.

1.5 Grading Student-Generated Flowcharts

The other portion of the computer-based education program attempts to "judge," to use the computer assisted instruction jargon, the flowchart as a correct solution to a specific word exercise.[3] In the literature, the two popular methods of grading programs are, first, to execute the program with randomly selected datasets [Barta and Nievergelt, 1975]; and, second, to derive verification conditions from the constraints of the input variables and the output variables, and to prove these verification conditions are true by a theorem prover [Manna, 1974]. Both has disadvantages in the computer-based education environment outlined above. The first cannot state whether a flowchart is correct, only that it found no error in the flowchart; the second method, an important research area in Computer Science, is not suitable because of the processing and storage constraints. These distinctions will be discussed in detail in Chapter Two. The Semantic Formulation Method (hereafter referred to as SFM), developed by the author and covered in detail in Chapter Three, tries to deduce equivalence between the student's flowchart and an instructor's flowchart. For

[3]"A specific word exercise" refers to a student homework assignment written in English, much like the algorithms found in Knuth's The Art of Computer Programming, Vol. 1 [Knuth, 1973].

SFM to prove correctness, the student's flowchart has to be moderately close to the instructor's. An interactive system, SFM can point out the problematic area in instances when it cannot prove correctness. SFM does not and cannot hope to compete in proving programs correct with the techniques of program verification of Manna, Hoare, et al., which utilize minutes of CPU time, thousands of words of storage, and special purpose languages, e.g., MICROPLANNER [Sussman, Winograd, and Charniak, 1971] or QA4 [Derksen, Rulifson, and Waldinger, 1972]. With the limited resources in a computer-based education environment, SFM can prove correct flowcharts of about 15 boxes using only a few hundred words of storage in less than one second of CPU time (one minute of real time).

1.6 Brief Description of Semantic Formulation Method (SFM)

In brief, SFM shows equivalence of a student's flowchart to the instructor's flowchart by logical deduction on a Semantic[4] Model. The program "SELECTOR" selects portions of the student's flowchart and the instructor's flowchart to be candidates as semantically equivalent pieces. A powerful "MATCHOR" checks for local consistencies. After all portions of the student's flowchart and the instructor's flowchart have passed scrutiny by the SELECTOR and

[4]"Semantics" in this sense refers to global semantics of artificial intelligence, not "semantics" in the sense used by compiler or programming language researchers.

the MATCHOR, the Semantic Model is formed. If the deduction on the Semantic Model hits an inconsistency, backtracking occurs and the program returns control to the SELECTOR to find new candidates. SFM will be discussed in Chapter Three; some of the detailed algorithms will appear in Chapter Five.

1.7 Research Methodology

Regarding the research methodology followed in this thesis, the author feels there are two general approaches, with shades in between, in pursuing research. The first general approach extends a theory and then looks for application areas to justify the extension. The second general approach attacks a problem in an application area by extending whatever theory or theories are appropriate. The author used the latter approach to pursue the general research problem: "How does one teach a beginning student to plan the flow of ideas in a computer program?" In the solution, theories in the areas of artificial intelligence, computer assisted instruction, and software engineering were extended.

The research has focused on designing a viable program to judge flowcharts in the above-mentioned computer-based education environment. Using the touch panel on the PLATO IV system, a student can "draw" a flowchart which he/she feels is representative of a specific word problem. The program should interact fast (under one CPU second or one minute of real time) to give the student useful feedback and should require only a few hundred words of data per student.

The following chapters will discuss the design of the program, which demonstrates the feasibility of machine grading student-generated flowcharts in a computer-based educational environment.

Chapter Two

REFINEMENT OF THE PROBLEM AND SURVEY OF THE RELEVANT LITERATURE

Chapter One presented an overview of the research effort. This chapter will refine the research effort and discuss various backgrounds in four research areas of computer science: software engineering, program verification, computer-assisted instruction (CAI), and artificial intelligence (AI).

2.1 Restatement of the Problem

The overall goal is to teach algorithmic development to beginning programmers. Arguments shown previously have led the author to decide to teach structured programming--with flowcharts--in a computer-based education environment. More specifically, a student (after several hours of instruction in structured programming) is presented with a word exercise and is asked to "draw" on the PLATO IV screen a flowchart which represents the word exercise. The Program to Analyze Smooth Flowcharts (PASF) should grade the flowchart, giving fast feedback to the student. As part of the structured programming technique, the student must draw a flowchart with a limited control structure, which is called a smooth flowchart by the author.

2.2 Smooth Flowcharts

The following sections cover the historical and theoretical background involved with the invention of smooth flowcharts. An informal discussion as well as the formal definition of smooth flowcharts is included.

2.2.1 Software Engineering

At the end of the Sixties, the computer industry began experiencing a software "crisis," i.e., discovery that software production is expensive. Major projects were budgeting money to software equal to the cost of the hardware. Even then, the production of the software often slipped past several deadlines and the final product was unreliable. To avert this crisis, the discipline of computer science formed an area of research called software engineering.

2.2.2 Structured Programming

So that software may be made more reliable, prominent computer scientists began to study how programmers program and to criticize the methods and tools available.

> I see great future for very systematic and very modest programming languages. When I say "modest," I mean that, for instance, not only ALGOL 60's "for clause," but even FORTRAN's "DO loop" may find themselves thrown out as being too baroque.[5]

---

[5]Edgsger Dijkstra, "The Humble Programmer," CACM, Vol. 15, No. 10, Oct., 1972, p. 865.

Dijkstra has not only criticized the familiar high level languages, but has proposed a well-thought-out alternative called structured programming [Dijkstra, 1970].

In his technique of structured programming, Dijkstra proposed a control structure which is more limited than ALGOL, FORTRAN, or other programming languages. The basic constructs are DOWHILE, IFTHENELSE, and an assignment statement. The structured programming technique is more than programming with only these constructs; it is a way of thinking about how to solve a problem [Gries, 1974].

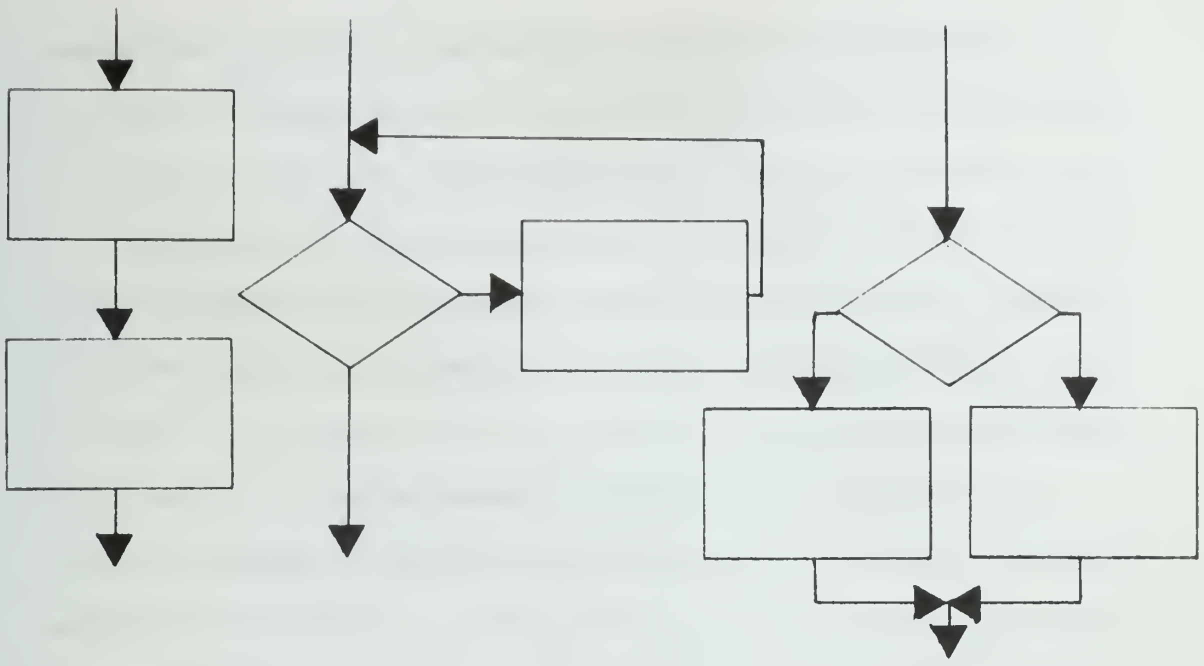
An integral part in this problem-solving process is the step-wise refinement technique [Wirth, 1971]. The step-wise refinement technique is basically a top-down approach to programming in which the programmer starts with the problem written in one sentence (usually in English, Dutch, etc.). He divides this sentence into a DOWHILE loop, as IFTHENELSE decision, or into several parts. At each step he refines the English phrases into programming code and more detailed English phrases. The programmer continues to divide and refine the portions until the program is finished. A. V. Aho calls the step-wise refinement technique "divide and conquer" [Aho, 1974].

Not only does Dijkstra feel that FORTRAN's DO loops are too baroque, he also considers the GOTO statement harmful [Dijkstra, 1968]. Although the GOTO controversy was already kindled, the above paper by Dijkstra fanned the coals of controversy into a conflagration. There is no need to reopen the controversy here, for it is excellently discussed in a paper by Knuth [Knuth, 1974]. The author agrees with Gries [Gries, 1974] and Denning [Denning, 1974] that the whole GOTO controversy misses the point. Structured programming's forte is not that it does or does not eliminate GOTO's, but that it is a disciplined way to solve problems. If, after solving the problem using the limited control structure, a programmer rewrites the program in FORTRAN with GOTO's, that is acceptable.

2.2.3 Informal Description of Smooth Flowcharts

The author proposes his own limited control structure called smooth flowcharts. They are a subset of all possible flowcharts, modeled after Dijkstra's DOWHILE, IFTHENELSE, and assignment statement. Smooth flowcharts allow any rectangle to be replaced by one of the following three constructs as shown in Fig. 2-1.

Jacopini [Bohm and Jacopini, 1966] proved that these three constructs can form a restricted flowchart which is semantically equivalent to an arbitrary flowchart. At the time, this appeared to be an extremely important result. Lessening the importance, Cooper [Cooper, 1967] quickly responded that Jacopini used extra variables, and that if extra variables could be used, he had a scheme to reduce any flowchart to one loop with a variable used as a program counter. The theoretical question arose asking whether any flowchart can be redrawn by the above three constructs



1. A string of rectangles concatenation
2. A DOWHILE loop
3. An IFTHENELSE

Fig. 2-1. The Three Possible Constructs in a Smooth Flowchart

without the addition of a variable. Knuth and Floyd [Knuth and Floyd, 1971] proved it is not possible. Kasai [Kasai, 1974] proved that the necessary and sufficient conditions for a general flowchart are translated into the above constructs. But translating from general and perhaps confusing flowcharts to the above three constructs is not the point. As Dijkstra has pointed out,

> The exercise to translate an arbitrary flow diagram more or less mechanically into a jumpless one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.[6]

---

[6]E. W. Dijkstra, "Go to statement considered harmful," Comm. ACM, Vol. 11, No. 3, March, 1968, p. 538.

The author agrees with Dijkstra and others that a limited control structure should be taught as an integral part of structured programming. In PASF, the student will be taught smooth flowcharts in the process of learning the step-wise refinement technique. This is not a major hurdle because if a student follows the step-wise refinement technique, he automatically generates smooth flowcharts.

The author has termed his limited control structure "smooth flowcharts" to distinguish them from D-flowcharts (D is for Dijkstra. Sometimes they are referred to as D-charts.) in the literature. Smooth flowcharts are exactly D-flowcharts, except smooth flowcharts require only one "stop." This decision was based on pedagogical not theoretical grounds. The author's version of step-wise refinement requires the student to use only one "stop." The theoretical aspects of this paper can be extended easily without this restriction. Credit for the use of the modifier "smooth" is given to Professor Foulk of Ohio State University [Foulk, 1973], who talks about "smooth" programs. The term "structured flowcharts" was not used because this in the literature refers to any flowchart which can be parsed by a phrase-structured grammar [Arnborg, 1974].

2.2.4 Formal Definition of Smooth Flowcharts via Web Grammar

For a precise description of smooth flowcharts, the entities called web grammars are introduced. Grammars on strings are well-known [Hopcroft and Ullman, 1969]. For example, consider the grammar G_1 on (V_N, V_T, P, S). The symbols V_N, V_T, P, and S are the nonterminals, terminals, productions, and start symbol respectively. The grammar G_1, where $V_N = \{S\}$, $V_T = \{a\}$, and $P = \{S: = Sa, S: = a\}$, generates all the strings with one or more "a's."

Pfaltz and Rosenfeld [Pfaltz and Rosenfeld, 1969] introduced web grammars to allow one to generate classes of webs analogous to strings. A web is a directed graph with symbols at its vertices. Web grammars are in the form of (I, V, R) where I is the initial web, V is the vocabularies V_N and V_T, and R is the rewriting rules. For example, consider the web grammar $W_1 = (I, V, R)$ where the initial web, $I = \{\underset{\bullet}{S}\}$, $V_N = \{\underset{\bullet}{S}\}$, and $V_T = \{\underset{\bullet}{a}\}$ and R = $\{\underset{\bullet}{S}: = \underset{\bullet\!\longrightarrow\!\bullet}{S\ \ a}, \underset{\bullet}{S}: = \underset{\bullet}{a}\}$.

One further detail that is required to specify the web grammar is Pfaltz and Rosenfeld's idea of embedding (E). Each rewriting rule must explicitly state how the web part fits into the host web. For example, the following is the embedding for the first rewriting rule of W_1:

R1: $\underset{\bullet}{S}: = \underset{\bullet\!\longrightarrow\!\bullet}{S\ \ a}$ E = {(x, S) | (x, S) an edge in the host web}$\cup$
{(a, x) | (S, x) an edge in the host web}.

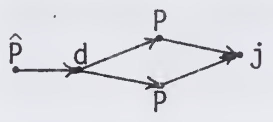
Web grammars, like string grammars, can be used for parsing or generation. Assuming one is parsing a web, the embedding for R1 means that any edge entering $\underset{\bullet}{S}$ will enter $\underset{\bullet}{S}$ after the replacement by the left side of R1, and any edge leaving $\underset{\bullet}{a}$ will be leaving $\underset{\bullet}{S}$. This is actually what one would expect. Montanari [Montanari, 1970] perfers to call this "normal" embedding and dispenses with the long set notation. The grammar W_1 will generate all the webs of the form

$$\begin{tikzcd} a \arrow[r] & a \arrow[r] & a \arrow[r] & a \arrow[r] & a \end{tikzcd}$$

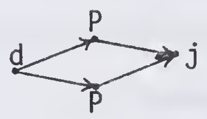
Below is a web grammar W_2 defining smooth flowcharts: the initial web, $I = \{s \rightarrow P \rightarrow e\}$; $V_N = \{\underset{\bullet}{P}\}$; $V_T = \{\underset{\bullet}{s}, \underset{\bullet}{e}, \underset{\bullet}{b}, \underset{\bullet}{d}, \underset{\bullet}{j}\}$

R1: $\underset{\bullet}{P} := P \rightarrow b$ $E = \{(x,P) \mid (x,P)$ an edge in the host web$\} \cup$
$\{(b,x) \mid (P,x)$ an edge in the host web$\}$

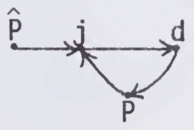
R2: $\underset{\bullet}{P} := \underset{\bullet}{b}$ $E = \{(x,b) \mid (x,P)$ an edge in the host web$\} \cup$
$\{(b,x) \mid (P,x)$ an edge in the host web$\}$

R3: $\underset{\bullet}{P} :=$ 

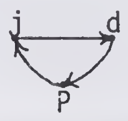
$E = \{(x,\hat{P}) \mid (x,P)$ an edge in the host web$\} \cup$
$\{(j,x) \mid (P,x)$ an edge in the host web$\}$

R4: $\underset{\bullet}{P} :=$ 

$E = \{(x,d) \mid (x,P)$ an edge in the host web$\} \cup$
$\{(j,x) \mid (P,x)$ an edge in the host web$\}$

R5: $\underset{\bullet}{P} :=$ 

$E = \{(x,\hat{P}) \mid (x,P)$ an edge in the host web$\} \cup$
$\{(d,x) \mid (P,x)$ an edge in the host web$\}$

R6: $\underset{\bullet}{P} :=$ 

$E = \{(x,j) \mid (x,P)$ an edge in the host web$\} \cup$
$\{(d,x) \mid (P,x)$ an edge in the host web$\}$

$\hat{P}$ is used to distinguish between Ps in the embedding if more than one P appears on the right-hand side of a rewriting rule. $\hat{P}$ has no other significance. The rewrite rules are drawn in such a way that the embedding is simply seen. All edges entering the leftmost node enter P; all edges leaving the rightmost node leave P. (This embedding is for parsing; generation is the reverse.)

In the above web grammar W_2, the nonterminal P and terminals s, e, d, b, j are mnemonics for process, start, end, decision, box (rectangle), and join. Below is a sample web.

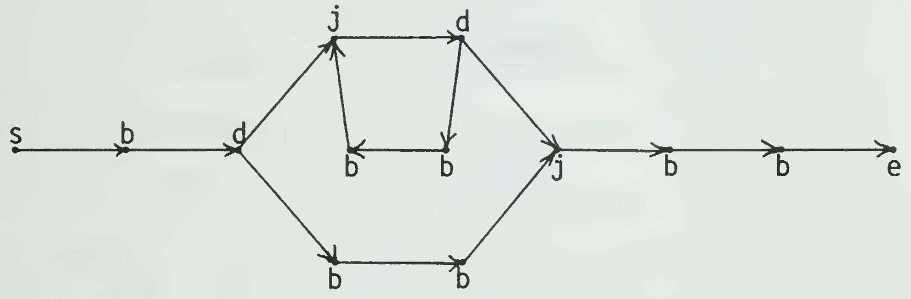


Fig. 2-2. Web A

By substituting ⬭ for s and e, ▭ for b, and ◇ for d, webs generated by W_2 can be redrawn in the standard flowchart notation. Smooth Flowchart A (cf. Fig. 2-3) has been drawn from top to bottom, whereas Web A (cf. Fig. 2-2) was drawn from left to right.

Since PASF requires that the student draw a smooth flowchart, an algorithm to check whether an arbitrary flowchart is a smooth flowchart must be included. The web grammar W_2 forms the

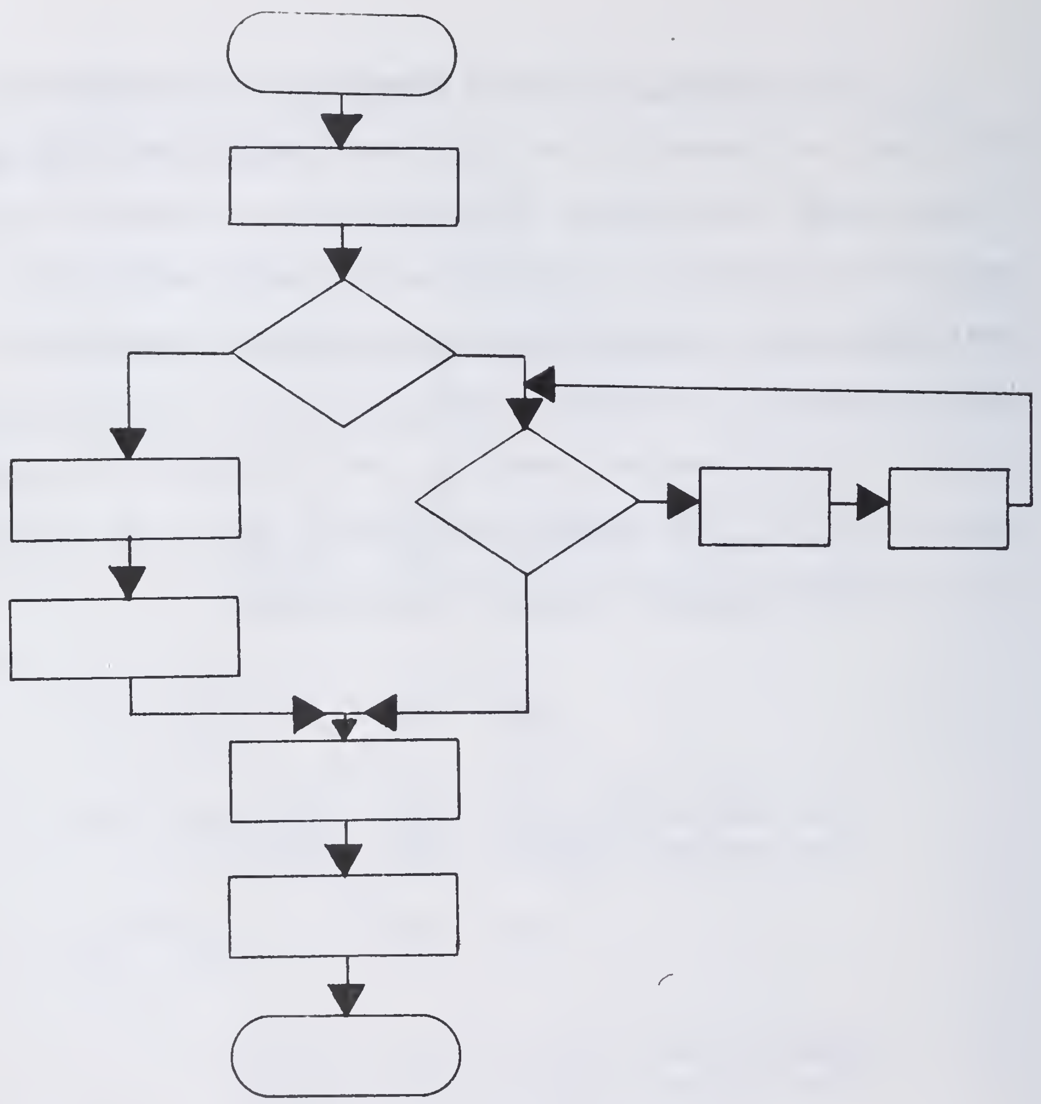


Fig. 2-3. Smooth Flowchart A

basis for this algorithm. In Chapter Five, a recursive algorithm based on the web grammar W_2 which checks whether a flowchart is smooth or not is described.

2.2.5 <u>Why Flowcharts and Structured Programming?</u>

Several people in the literature [Gries, 1975] have stated that flowcharts should _not_ be used in conjunction with structured programming. These opponents to flowcharts say that one rationale for using structured code is for readability, and that flowcharts allow a student to draw a rat's nest of boxes and arrows. The author concedes that the opponents have a good point. Thus, students should be shown the need for neatly drawn flowcharts. ([Nassi and Shneiderman, 1973] have devised a technique for "cleaner" flowcharts for structured programming.) In spite of this deficiency in readability, the author believes that flowcharts are beneficial to beginning programmers, especially nonmajors. That many people think in graphic terms, as opposed to verbal terms, is one reason that flowcharts are easily comprehended by beginners. Another reason is that, since almost all fields of learning have representations with boxes and arrows, many nonmajors find security in using flowcharts. Finally, flowcharts were incorporated for reasons of compatibility; flowcharts, like FORTRAN, are known and understood by many programmers.

2.3 Grading Student-Generated Flowcharts

Ever since the training of programmers began, instructors have wanted to be able to grade programs by machine. This section discusses three methods of automatic grading: test data, inductive assertion, and the author's own SFM.

2.3.1 Grading by the Test Data Method

Automatic grading programs appeared in the late Fifties [Forsythe and Wirth, 1965] as soon as high-level languages became widespread. These early automatic graders [Naur, 1964] embed the student's programs within a larger grading program which supplies test cases and performs checks and evaluations on the output variables. Randomized datasets were used in the test cases to discourage the students from cheating. Although barely mentioned in the literature ([Barta and Nievergelt, 1975] is an exception), this technique is heavily used in programming courses on many campuses. Many automatic graders perform other tasks as well as check the student's output variables against the right answers. Different features include monitoring the number of iterations for convergence of numerical analysis algorithms, checking execution time, checking the order of the output variables, checking the tolerances on the output variables, and assigning grades to the student. This style of automatic graders suffers from major drawbacks. The first drawback is the numerical analysis problem, e.g., computers compute different numerical answers to the same expression. (Expressions A (B + C) and AB + AC are not necessarily equal because of roundoff error and imprecise numerical representation.) The second and more important drawback is that this technique does not rely on or use the structure of the student's algorithm. "Good" coding and efficiency are related to the logical structure of the program.

Furthermore, the test data method can never say that a student's program is correct, only that the student's program has no detected errors.

2.3.2 Grading by the Inductive Assertion Method of Program Verification

Program verification by the inductive assertion method can prove programs correct. The ideas for this approach started with Von Neumann and Turing [Manna, 1974]. Floyd [Floyd, 1967] suggested that a program could be proved correct by attaching assertions about the values of the variables at different points throughout the program and by proving that the assertions hold true for all computations of the program. The inductive assertion method (A good introduction is in [Gerhart, 1975]; it is also well covered in [Manna, 1974].) consists of four steps. Step 1 is to breakup the flowchart program into what Manna calls cutpoints. One cutpoint is at the start, another is at the end of the program, and at least one is within every loop. An assertion (Step 2) is placed at every cutpoint. The assertions at the loop cutpoints[7] are the most interesting and difficult, since they must reflect the induction character of the loops. For every path from one cutpoint to another cutpoint (with no cutpoints in between), a verification condition is constructed (Step 3). The final step is to prove all

[7]In the literature this is also referred to as the loop invariant. There is a strong movement [Gries, 1975; Adams, 1975] to teach programmers to think and code in terms of loop invariants.

the verification conditions are true. If all four steps are completed for a given program, then the program is partially correct with respect to the input and output assertions. To prove the program is totally correct, the program must be partially correct and be shown to always terminate. A program always terminates if it can be shown that there are no infinite loops for any set of values of input variables. Proving programs partially correct by the inductive assertion method is very promising, but, at the moment, only medium-sized programs, e.g., FIND [Hoare, 1971], can be proved. Many of the proofs of programs in the literature were done manually using this method [London, 1972]. Extensive efforts to automate the method are underway. Researchers have shown that Steps 1 and 3 are mostly mechanical. To prove the verification conditions of Step 4, researchers (e.g., [Igarashi, London, and Luckham, 1973]) are utilizing the power (and problems) of automatic theorem provers. The researchers' major hurdle in automating the method is discovering the appropriate set of inductive assertions (Step 2). Most automatic program verification systems have the user insert the inductive assertions. Several heuristic techniques have been developed [Wegbreit, 1973; Katz and Manna, 1973] to allow automatic systems to attempt to generate the inductive assertions. Why is finding the inductive assertions for loops so difficult? Proving that a loop is correct is closely akin to an induction proof in mathematics. Research with automatic theorem provers has shown that induction proofs are extremely difficult. Thus, it can be assumed that finding loop assertions will continue to be a major hurdle for automatic inductive assertion systems.

Although the inductive assertion method is very promising for reliable software, there is little future in using it as a grader in the previously discussed computer-based education environment. Aside from the fact that any automated version of the inductive assertion method would need thousands of words of core, minutes of CPU time, and a special AI language, the method would be inadequate on further grounds. Envisioning the use of the inductive assertion method as a grader, the ideal situation would allow the instructor to specify the input and output assertions, with everything else automatic. However, as discussed above, the state of the art has not reached this ideal. Using the current state of the art, not only must the instructor supply the loop assertions, but also the student must append assertions to his/her version of the program. Here arises a philosophical question: Should programmers be taught from the beginning to understand and use loop assertions (also referred to as loop invariants)?[8] As noted in Footnote 7, there is emerging a strong faction of computer scientists who feel that this should be done. Gries [Gries, 1975] speaks of this as being the creative part of programming. On the

---

[8]Clearly, one does not expect programmers to write assertions in higher-order predicate calculus, but perhaps they could write assertions much like the assert statement of ALGOLW.

other hand, one wonders whether it is worth changing the lives of many programmers for a specific program correctness method which may only be a fad. For example, there are other methods of proving correctness [Manna, Ness, and Vuillemin, 1972]. From the standpoint of economics, the proponents of teaching loop invariants state that programmers should manually prove programs correct in order to improve reliability and lower the development cost of software. Some are skeptical of teaching loop invariants and program correctness to programmers, because they are not convinced that the programmers will be able to apply the techniques to large programs, e.g., operating systems. There has been some success in teaching this technique of loop invariants and program correctness in advanced programming courses [Gries, 1975], but there have been mixed results in teaching it in beginning programming courses [Gerhart, 1975; Adams, 1975]. The author believes that beginning programmers, especially nonmajors, cannot be expected to state the inductive assertions.

2.3.3 Comparison of SFM with Test Data and Inductive Assertion Methods

In this section, two popular approaches to grading, executing random test data and the inductive assertion method, will be compared with SFM. These comparisons of the three methods are summarized in Table 1. Both of these popular methods have limitations, regardless of the user environment. The test data method

Table 1

Comparison of the Two Popular Grading Methods with SFM

Issues	Test Data	SFM	Inductive Assertion
1. Instructor Inputs	Sets of input values and corresponding output values	Typical correct answer and the answers to questions	Input and output assertions (loop assertions)
2. Student must write assertions?	No	No	Yes
3. Numerical analysis problem on correct output values?	Yes	No	No
4. Student errors	Compile errors: student output not equal to correct output values	Locates specific problem and points out where the error occurred	Could find, but usually only correct or not correct
5. Can detect the structure in the student's answer	No	Yes	Yes
6. Can determine efficiency of student's answer	Little	Some	Some
7. Storage required	Little	Couple hundred words	Thousands of words
8. CPU power required	< sec.	About 1 sec.	Minutes
9. Power of method	?, no	Correct, ?, no	Correct, ?, no

cannot be used to show correctness; the inductive assertion method requires the student to supply assertions about his/her program. Further limitations are imposed when a user environment is selected. Assuming that the user environment is a computer-based education one, the test data approach is weak in diagnosing the student's logical and semantical errors. The heavy machine constraints of a computer-based education environment, e.g., computational power and storage, are extremely restrictive for the inductive assertion method. In spite of these drawbacks, the test data approach is being used in computer-based education systems [Barta and Nievergelt, 1975]. In the hope of discovering powerful theoretical tools, there is renewed interest in the test data approach for testing and validation of programs [Gerhart, 1975; Osterweil and Fosdick, 1974]. To alleviate some of the problems imposed by the above limitations, this thesis proposes an alternative method for grading student-generated programs.

The author's SFM (Semantic Formulation Method) has some of the advantages of both of the popular methods, but is restrictive in other senses. First, SFM has an advantage over the test data method in that SFM can prove some programs correct. Again, this is only partial correctness. SFM does not consider the total correctness problem, i.e., does the program terminate? The range of programs which SFM can prove correct is much smaller than the range of the inductive assertion method, but an advantage over the inductive assertion method is that SFM does not need assertions from the student, only the student's program. Concerning the issue of student errors, SFM's main emphasis is on finding student errors and pointing out where the problem lies. Since the test data method does not concern itself with the structure of the student's program, the test data approach is very limited in this regard. The inductive assertion method could be adopted to concern itself with student errors, but the author knows of no research which concerns itself in depth with this issue. One major advantage of the test data approach is that it uses little computation time and storage. SFM also has this advantage, and uses little computation time and storage, whereas the inductive assertion method typically uses a lot of time and memory. While both the test data and the inductive assertion methods may manage with only the input and output assertions, SFM must have an idea of the internal structure of the student's algorithm, i.e., the instructor must input a correct program. This means that both of the popular methods could grade a class of sort routines, but SFM's performance would be limited. Since SFM requires the students to input the same algorithm as the instructor's, SFM can compare two bubble sorts but not two arbitrary sort routines. Within the above-mentioned limitations, SFM can prove a small domain of programs correct in a computer-based education environment.

2.4 Equivalence of Programs

The question of whether two programs are equivalent has been asked by computer scientists for a long time. Many areas of computer science, e.g., compiler optimization [Allen, 1969] and parallel processing [Bernstein, 1966], have searched for an answer to this question. Turing [Turing, 1936] and others proved that, in general, determining whether two programs are equivalent or not is undecidable (the two programs may never halt). Realizing this, researchers turned to the more restrictive problem of equivalence of "program schemas" which takes into account less of the semantic structure of the program [Keller, 1971; Aho and Ullman, 1970]. This research, not a program schema approach, has restricted the programming domain rather than abstracting to program schemas. It is hoped that the programming domain is sufficiently small to determine a large percentage of the time whether two programs are equivalent.

In the literature, there are several different definitions of equivalence. Two are input/output equivalence and computational equivalence. The former asks only that the two programs give the same output for an arbitrary input. A bubble sort program and a ripple sort program could be shown to be input/output equivalent. The inductive assertion method of program verification assumes input/output equivalence. Computational equivalence requires that both programs contain identical computational sequences [Lee, 1972].

2.4.1 Global Semantic Equivalence

The author's Global Semantic Equivalence (GSE) is a mixture of input/output equivalence and computational equivalence. At a local level, SFM tries to find pieces of the student's program and the instructor's program which are input/output equivalent. At a global level, SFM searches for computational equivalence between the pieces shown to be input/output equivalent. What a piece is and how big a piece is is dependent on the intelligence of the AI program PASF and the combinatory problems of comparing the pieces. Since a more intelligent PASF would require a combinatorial increase in comparisons, a compromise must be made in the design of PASF. Furthermore, SFM allows commuting of the pieces of the student's program which are independent [Bernstein, 1966]. GSE between the extremes of input/output equivalence and computational equivalence allows PASF to say that a student's program is correct and diagnose student errors in a reasonable amount of computation.

If all the pieces of the student's program were _individually_ semantically equivalent to the pieces of the instructor's program, the student's program could still be incorrect. The notion of the whole program being equivalent must be included in Global Semantic Equivalence (GSE). For a student's program to be Global Semantically Equivalent to the instructor's program, all the pieces of the student's program must fit together in a manner similar to the pieces of the instructor's program. This manner is related to the word exercise. If each piece of the student's program corresponds to a part of the word exercise which corresponds to a piece of the instructor's flowchart, the two pieces are GSE. An example will demonstrate this notion of GSE. A word exercise for a student begins, "Read the ID number off a card. . ." If it can be determined by heuristics, analysis, or whatever means that the student's "read zap" refers to the reading of the ID number and that the instructor's "read a" also refers to the reading of the ID number, then these two portions of a program are GSE. If all of both programs can be shown to be consistent, corresponding to portions of the word exercise, then the two programs are GSE. Notice that this use of the term "semantics" is different from the normal use in programming languages or compiler construction. Both "read a" and "read zap" have the same "semantic action" in that they both reference an input device to assign a value. The "read a" and the "read zap" need not be GSE, depending on the rest of the two programs. There may be more than one "read" in both programs. Thus, the instructor's "read a" may be GSE to "read zap" but not GSE to "read cow" in the student's program. How variables "zap" and "cow" are used later in the program will determine whether "read a" is GSE to "read zap" or "read cow." This idea of semantics concerns the whole program; hence, the use of the modifier "global."

In this conception of semantics and equivalence, three entities are important: the word statement of the exercise, the instructor's flowchart (and, as will be shown later, other information from the instructor), and the student's flowchart. Ideally, an artificial intelligence (AI) program should need only two: the word exercise and the student's flowchart. This is what is referred to in the AI literature as the "natural language problem" [Winograd, 1971], which is exceedingly difficult and is not attacked in this thesis. With the introduction of the instructor's role, the author hopes to overcome the "natural language problem." It is assumed that the instructor's flowchart is a correct formulation of the word problem. It is also assumed, but need not necessarily be true, that the same person as instructor writes the word exercise and inputs the correct answer via a flowchart. It is the task of the instructor to make the word exercise and the instructor's flowchart GSE.

The student's task is to "draw" a flowchart which is GSE to the word exercise. It may or may not be GSE to the word exercise. After the student has drawn the flowchart and submitted it for scrutiny by the grader, the grader checks to see whether the student's flowchart is GSE to the instructor's flowchart. If it is GSE, then a major premise of the thesis is that one can conclude that the student's flowchart was indeed GSE to the word exercise.

In Fig. 2-4 the model is incomplete because of the human processes involved. In this thesis, there is no attempt to explain or model these human (i.e., student or instructor) processes.

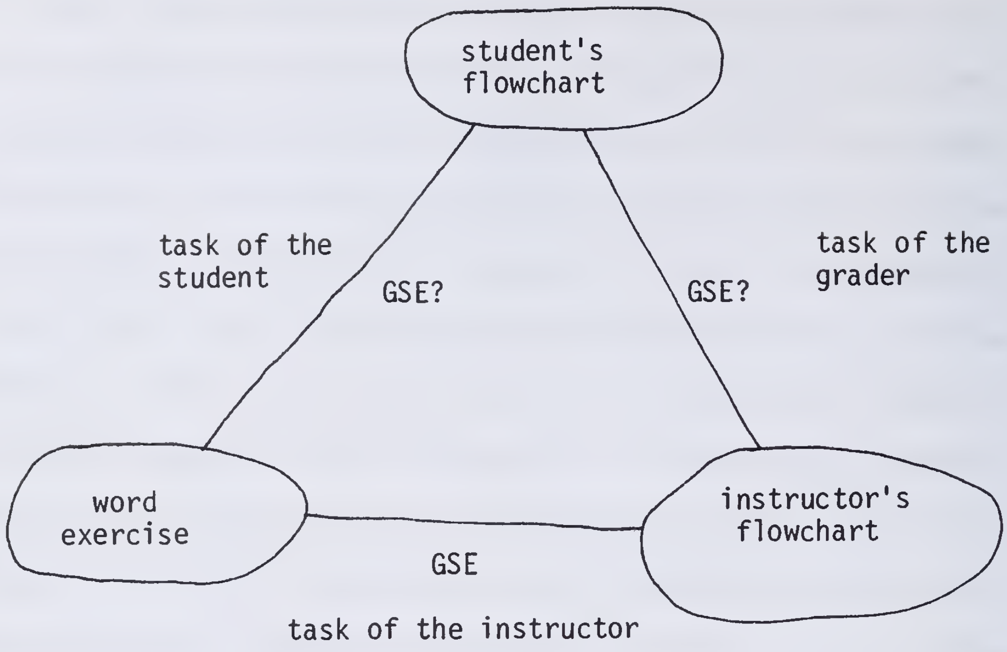


Fig. 2-4. Three Entities of the GSE Model

Although these processes are not formally known, everyone has a fair idea of what is meant by the student's task: i.e., the student should draw a flowchart which "represents" the word exercise. A fuller notion of the grader's task of trying to show GSE, as attempted by SFM, will be presented in Chapter Three.

A second observation about the GSE Model is that the grader may conclude that the student's flowchart and the instructor's flowchart are not GSE. If this is true, the student's flowchart could still be GSE with the word problem. Hopefully, the possibility that this will occur is minimized. Since GSE involves three entities, one must always speak of GSE in terms of all three.

This view of semantics as reflected in GSE was expanded from the field of AI. Winograd [Winograd, 1971] has been the most influential in the matter of semantics. Winograd speaks of a practical view of semantics:

> In practical terms, we need a transducer which can work with a syntactic analysis, and produce data which is acceptable to a logical deductive system.[9]

2.5 Artificial Intelligence Methodology

Much of the methodology in this thesis is related to methodology used by researchers in the area of AI. The author's thinking is strongly influenced by Minsky at MIT and his students Winograd, Hewitt, Charniak, and Sussman [Minsky, 1968; Minsky and Papert, 1972; Winograd, 1971; Hewitt, 1971; Charniak, 1972; Sussman, 1972]. Also influential were other major AI projects at Carnegie Mellon [Newell and Simon, 1972], Stanford Research Institute [Nilsson, 1971], Stanford [Derksen, Rulifson, and Waldinger, 1972; Waldinger and Levitt, 1974], University of Illinois [Ray and Preparata, 1972; McCormick, Ray, Smith, and Yamada, 1965; Biss, Chien and Stahl, 1971; Henschen, 1971], all in the United States of America, and Edinburgh in Scotland [Michie, 1967].

For years, the AI field followed the syntax-semantics paradigm. Minsky [Minsky and Papert, 1972] has proposed a new

[9]Terry Winograd, Procedures as a Representation for Data in a Computer Program for Understanding Natural Language, MIT, MAC TR-84, 1971, p. 280.

paradigm of "description, representation, and deduction." This latter paradigm has been followed in this work. AI techniques such as heuristic search, backtracking, and tree pruning are incorporated into this research. Further discussion of the AI techniques and design issues of SFM will appear in Chapter Six.

2.6 Grader vs. Tutor

Since ancient times, the ideal method of instruction has been to have one expert for every student, a one-to-one situation often called tutoring. In a computer-based education environment, machine tutors as well as machine graders are desirable. The distinction between a tutor and a grader in some cases may be slight; in others, the two are definitely different. A tutor is an expert in a certain subject matter domain. He/she/machine must be able to ask questions about the subject matter to direct the student's learning; furthermore, the student may need to ask questions of the tutor. Several attempts at machine tutors exist [Carbonell, 1971; Danielson and Nievergelt, 1975; Mateti, 1975]. This research is not a tutor but a grader. A grader only attempts to follow a grading procedure dictated by an instructor. A grader need not know the subject matter in great depth. For example, SFM does not have knowledge stored about sorting or searching algorithms, but rather has a general knowledge about algorithms.

2.7 Summary of Research Methodology

Teaching beginning programmers how to form "good" flowcharts has initiated research in several fields. From the field of software engineering has come the concepts of teaching the student and the design of smooth flowcharts. The vehicle for teaching the students, i.e., a computer-based education system, is from the field of computer assisted instruction. The philosophies of AI have helped to design and create the program which implements the ideas of SFM. This thesis deals with an AI program in a CAI environment which teaches concepts in the field of software engineering.

Chapter Three

PROGRAM TO ANALYZE SMOOTH FLOWCHARTS (PASF)

This chapter discusses the implemented computer program PASF (Program to Analyze Smooth Flowcharts). The organization of this chapter corresponds to the steps followed by a student as he/she constructs and checks his/her flowchart.

3.1 Anticipated Student Population Using PASF

PASF is a series of CAI lessons on the PLATO IV system. The target population using PASF are students in an introductory nonmajors computer science course (CS100-level). Typically, the students are sophomores in engineering, but they may be freshmen to graduate students in majors as diverse as English and animal science. The prerequisite for these courses is essentially a high school diploma. The students are expected to have algebra but not calculus.

3.2 Use of PASF in a Course

PASF is designed to be used in the first two weeks of a CS100-level course as a supplement to lectures. PASF is not oriented toward any single language or textbook; it can be incorporated with any algorithmic high-level language such as BASIC, FORTRAN, ALGOL, or PL/1. Any number of students may use PASF simultaneously. The availability of PLATO terminals with touch panels is the main constraint. To finish the material in PASF, a student takes a total of two or three hours of terminal time. The first section of PASF contains about an hour of instruction. This is followed by a laboratory in which the instructor can give as many exercises as he wants. Completing five exercises may consume the better part of the student's hour. To discourage students from cheating, PASF is organized to allow the instructor to change or create new exercises as often as he wants, e.g., every term. Further discussion of the instructor's role in PASF appears in Chapter Four.

3.3 Instructional Portion of PASF

The instructional portion of PASF is a conventional CAI tutorial of about fifty frames. Four topic areas are presented in text form, richly mixed with questions for student interaction. The four areas are

1. Definition and importance of algorithms,
2. The step-wise refinement procedure of structured programming,
3. Description and use of flowcharts, and
4. A combination of step-wise refinement and flowcharts to develop smooth flowcharts.

There is a quiz at the end of each topic. The student may review any previous frame if he/she so desires. This portion gives the student the concepts and tools necessary to try the program laboratory section.

3.4 Program Laboratory of PASF

The student is asked to "draw" on the PLATO screen smooth flowcharts corresponding to a series of exercises. A short introductory handout explains the operations of the program. First, the student types in the name of the algorithm to identify to PASF which algorithm he/she is attempting. Next, the student "draws" the flowchart and then asks for it to be checked. PASF, analyzing the flowchart, says it is correct, or maybe correct, or points out an error. The following sections of Chapter Three will discuss in detail the program laboratory of PASF.

3.5 Input Section of PASF

To input the flowchart, the student uses both the PLATO IV keyboard and touch panel which fits over the plasma panel of the terminal. The touch panel allows a program to respond when a person touches one of 256 regions (16 x 16) on the terminal screen. For a viable man-machine interface in the input section, almost all the interactions by the student are done by touching. There are only two times when the student must interact by typing: when he/she enters the name of the algorithm and when he/she enters text inside a box. Figure 3-1 shows a copy[10] of the screen after a

[10]PLATO IV has the ability to produce hard copy of the screen by means of an optical-xerographic copier by Varian. The copy is about 7/8 the size of the original screen.

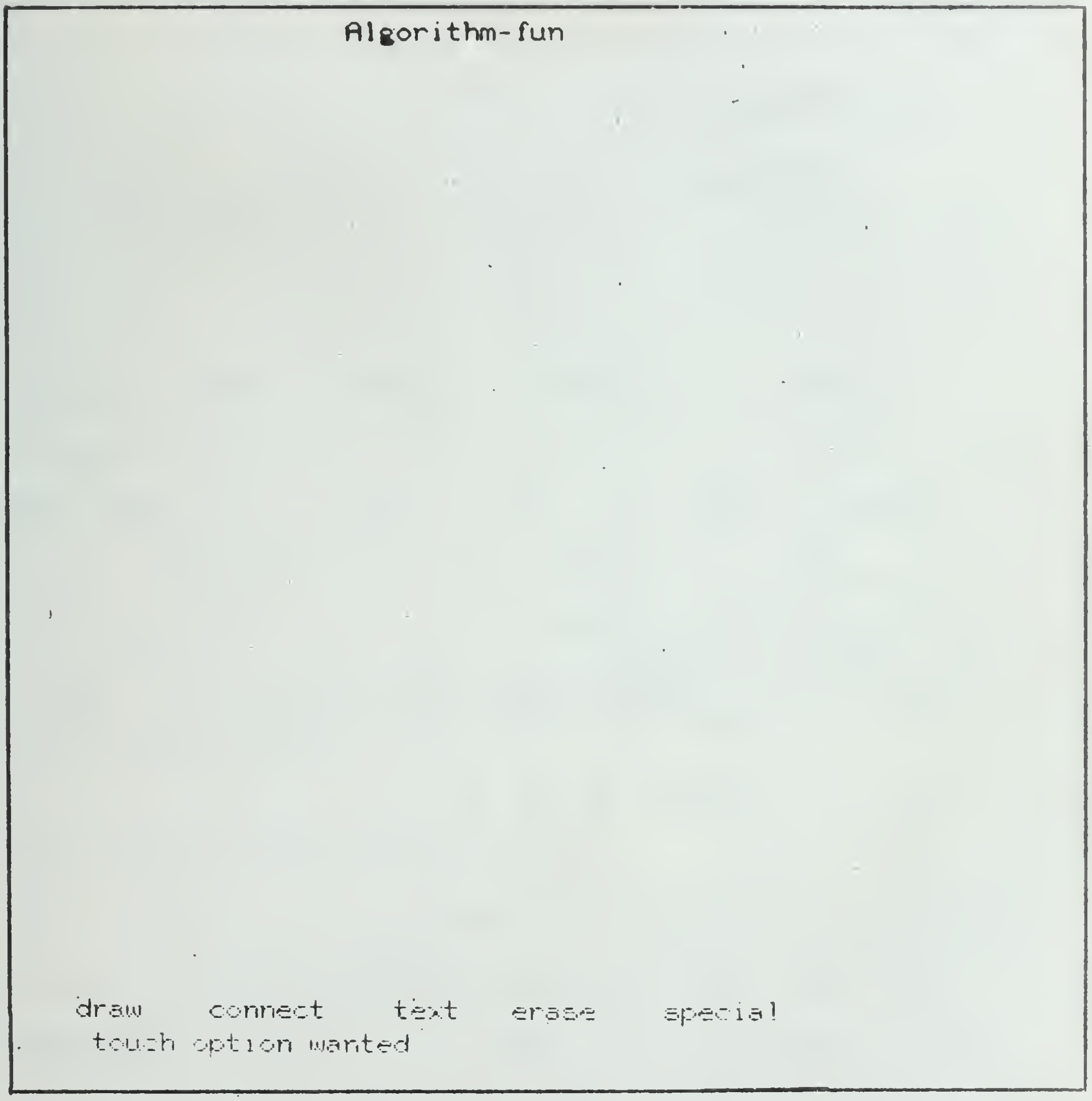


Fig. 3-1. Copy of PLATO IV Screen Showing Input Options

student has elected to work on algorithm "fun," but has not started to "draw."

3.5.1 Input Options of PASF

The chart below summarizes all the "touch" options of PASF.

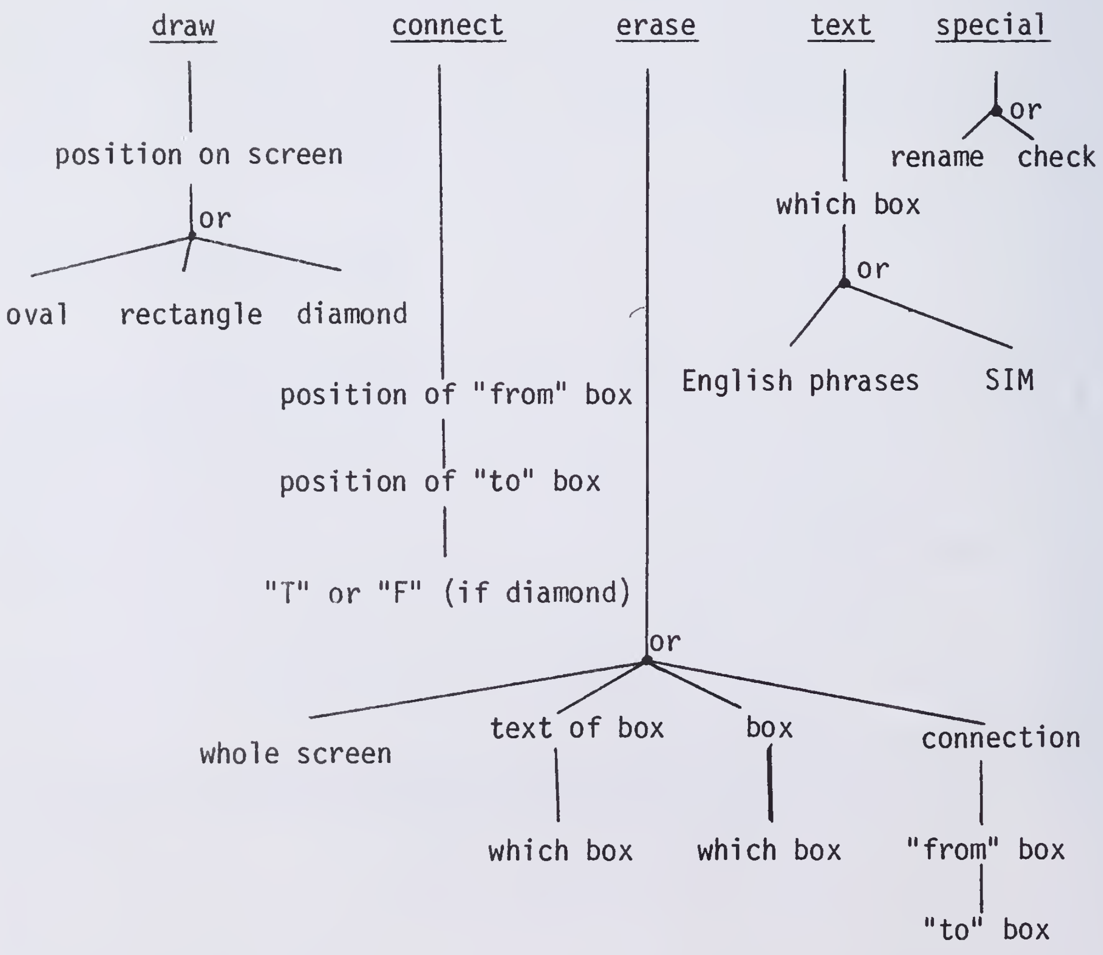


Fig. 3-2. Chart of Options to Touch

3.5.2 Free vs. Structured Approaches of Inputting

The author felt that a student should be allowed a free hand in creating the boxes, arrows, and text in any order that makes sense. The student does not have complete freedom, e.g., he/she cannot connect two nonexisting boxes. If a student tries something similar to this, an appropriate error message is given. Another possible approach would have been to force the student to draw the flowchart in a highly restricted fashion and check the flowchart at every step. This latter approach was not taken, since PASF is an attempt to model a grader, not a tutor. In a traditional course, the student would finish his/her flowchart before submitting it to a human grader; therefore, it is the responsibility of the student to decide when to submit his/her flowchart for checking by PASF.

3.5.3 Fast Feedback on Syntax Errors

One of the goals of PASF is to give fast feedback to the student. Instant feedback is given to the student when he/she types in the text of a box. A syntax check is made on the programming language inside the boxes as soon as the text is written. If a syntax error is found, the student is given feedback and PASF forces him/her to correct it. In addition, when a student types in an English phrase, it must match "closely" to the English phrases which have been allowed for that exercise and dictated by the instructor. There will be more discussion of English phrases later.

3.5.4 Experiment to Test Utility of Input Routine of PASF

To demonstrate that the inputting of the flowchart was reasonably fast and easy, an experiment was performed with several students. At one sitting, the student was asked to copy a flowchart from paper using pencil, paper, and an IBM flowchart template. At another sitting, the student was asked to input the same flowchart using PASF. Both situations were timed. It should be noted that the participants were noncomputer people and that they had a short training period using PASF. The results were that sometimes PASF was slightly faster (.9 the time for paper and pencil), but usually slightly slower (worst case being 1.5 the time for pencil and paper). The average is about 1.2 times longer for PASF. Observing the experiment, the author found that the longer times for PASF involved the student making a syntax error and PASF forcing him/her to correct it. The times for the paper-pencil cases does _not_ include any grading. For the paper-pencil case, some students did hand in flowcharts with syntax errors, e.g., an arrow missing or text missing in a box. Therefore, the average of 1.2 should be adjusted more toward 1.0, because PASF performs this extra task of grading. One must also remember that the time spent copying a given flowchart is a small fraction of time compared to the time required to construct a flowchart from a word problem.

Thus, the experiment proved that the man-machine interface of PASF is viable.

3.5.5 List Structure Representation of Student's Flowchart

As the student constructs his/her flowchart on the screen, PASF constructs a list structure representation internally. If the student adds or deletes a box, a node in the list structure is created or destroyed. A node consists of INFO for information, a left link (LL), and a right link (RL). (Cf. Fig. 3-3.)

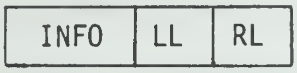


Fig. 3-3. A Node in the List Structure

As two boxes are connected by an arrow, a link in the list structure is added. RL is used for any arrows from oval or rectangular boxes and by the true side of a diamond box. LL is used for the false side of the diamond box. INFO includes the type of box, position on the screen, the text in the box, and several flags (used during checking).

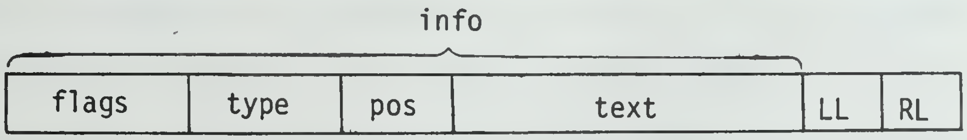


Fig. 3-4. INFO is made up of 4 Subfields

Enough information is stored in the nodes to completely replot the screen and regenerate the flowchart. On a full graphics terminal, screen dynamics are important and should be an integral part of the design.

Any computer-based education system must allow a student to leave the terminal and return, for example, a week later at a point very close to his/her exiting point. The list structure representation allows a student to leave and return with the same array of boxes and arrows. Another feature of PASF is that the student at any time may review the instructional portion of PASF. For example, if he/she has forgotten a rule of what contitutes a legal flowchart, he/she need only press a few keys to review the rules. After reviewing, the student hits a special key and his/her array of boxes and arrows is replotted.

3.5.6 Layout of Boxes and Arrows by PASF

Since the area on the terminal's screen is finite, the number of flowchart boxes plotted at any one time is limited. Because it was felt that switching between pages of portions of a flowchart would be confusing to a student and a hindrance to his learning, the author decided that the whole flowchart should appear and stay on the screen. (The only exception to this rule occurs when the student asks for remediation.) Therefore, the size of the boxes is relatively small to allow a fair number of them to appear on the screen. Twenty-one boxes can fit on the screen in three columns of seven.

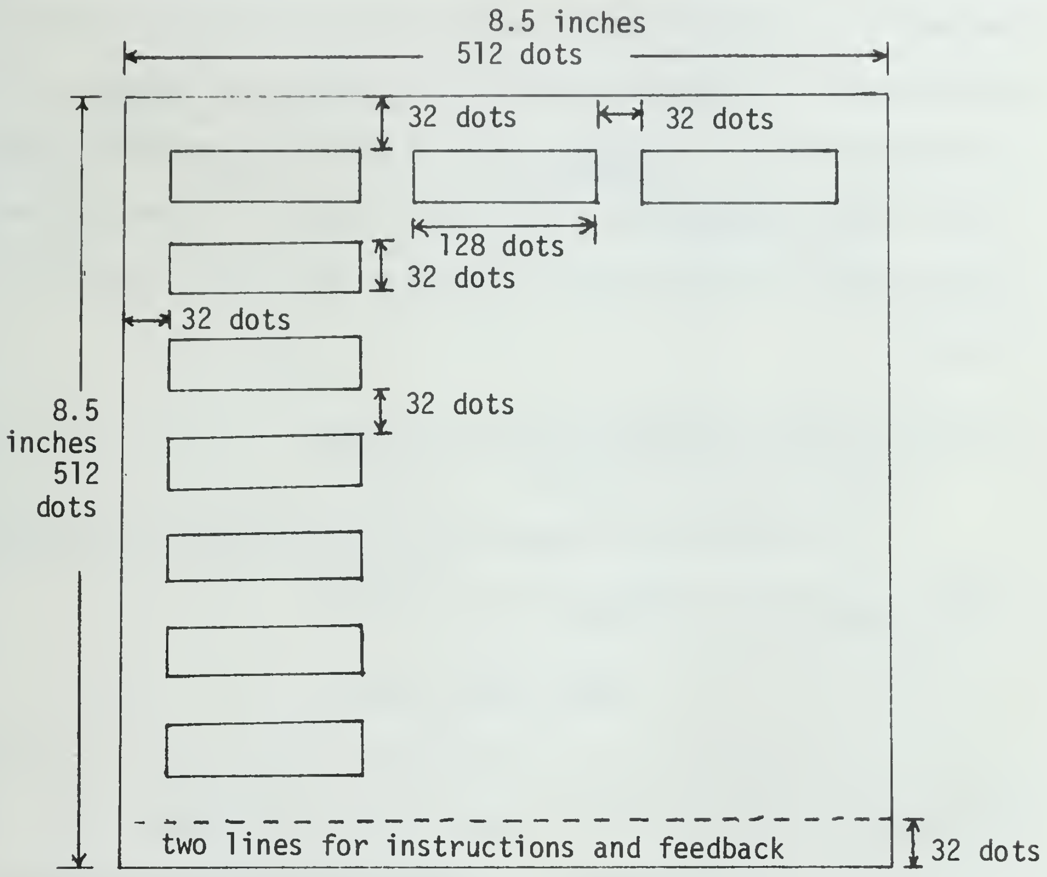


Fig. 3-5. Arrangement of Boxes on PLATO Screen

A rectangular box is 128-dots long and 32-dots high and allows two lines of 16 characters each of text. (There are 512 x 512 dots on the whole screen.) The author decided to make the three boxes the same size--each is 128-dots long and 32-dots high. Having the boxes the same size greatly facilitates constructing the arrows from one box to another. When a student connects two boxes, PASF usually draws an arrow from the middle of the bottom of the first box to the middle of the top of the second box. Since PASF automatically calculates the position and route for the arrow (cf. Section 5.2), the student need not worry about such details. For neatness, the author decided to plot the boxes in specified locations of rows and columns on the screen. When a student touches the screen to draw a box, the box is drawn at the closest specified location.

3.5.7 <u>SIM--the Programming Language of PASF</u>

The programming language SIM allowed inside the boxes is simple. Below is a Grammar for the language.

Productions of SIM

```
<statement> : = <oval stat> | <diam stat>   <rect stat>

                  $$ for ovals, diamonds, and rectangular-shaped
                  boxes respectively.
<oval stat> : = start | stop
<diam stat> : = <var con> <rel op> <var con>
<rect stat> : = <assign> | <input> | <output>
<assign> : = <assign 1> | <assign 2>
<assign 1> : = <var> ← <var con>
<assign 2> : = <var> ← <var con> <arith op> <var con>
<input> : = read <var>
<output> : = print <var>
<rel op> : = = | ≠ | < | > | ≤ | ≥
<var con> : = <var> | <con>
<arith op> : = + | - | x | ÷
<var> : = <letter> <alphanum> | <letter> $$ up to 10 characters maximum
<alphanum> : = <alphanum> <letdig> | <letdig>
<letdig> : = <letter> | <digit>
<letter> : = a | b | c | ••• | y | z
<digit> : = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
<con> : = <sign> <num> | <num>
<num> : = <digs> • <digs> | <digs> • | • <digs> | <digs>
<digs> : = <digs> <digit> | <digit>
<sign> : = + | -
```

Briefly, SIM allows

1. "Start" or "stop" in oval boxes,
2. Expressions like "A ≥ 10.0" in diamonds, and
3. "Read cat," "print dog," and "dog ← -12 ÷ cat" in rectangles.

Notice that there are no IF, GOTO, or DO statements. The control structure is the smooth flowcharts. Smooth flowcharts allow DOWHILE and IFTHENELSE constructs.

To simplify the programming domain for the student (this is the student's first or second week in the course), there are no FORMATs for input/output, no arrays, and no procedure calls. Data types other than real variables and constants are not allowed. The hope is to have a simple programming language which the introductory nonmajor student can master quickly. For example, this language does not have a hierarchy of operators in expressions, which introductory students often find confusing. In SIM, the student must use several statements to accomplish the same expression. This does lead to inefficient use of temporary variables, but this is not considered detrimental, since the main concern is to have the student write a correct solution and not to achieve efficiency. Use of expressions with boolean operations, e.g., "a = b AND z = 10" is also not allowed. This is not a drawback, for the "AND" can be handled by the control structure.

One rationale for a programming language with short statements is that the space inside the boxes (maximum of 16 characters) is limited. To advanced programmers, SIM may appear too restrictive, but for the class of problems and the anticipated audience, it is adequate.

3.5.8 Intdivide--a Sample Flowchart

Fig. 3-6 is an example of a ten-box flowchart of an algorithm which does an integer divide by successive subtraction.

3.5.9 English Phrases inside Boxes

The domain of the allowable student programs is any smooth flowchart of twenty-one boxes or less containing SIM. One other feature, English phrases, is also in the domain. The intended use of PASF is to teach step-wise refinement; therefore, English phrases must be allowed inside diamonds and rectangles. Since PASF needs to distinguish between SIM and English phrases, the student specifies whenever he/she wants to type in an English phrase. The English phrase must match what the instructor has designed to be permitted for a specific exercise. A discussion of this restricted natural language feature and how it is implemented will appear in Chapter Four.

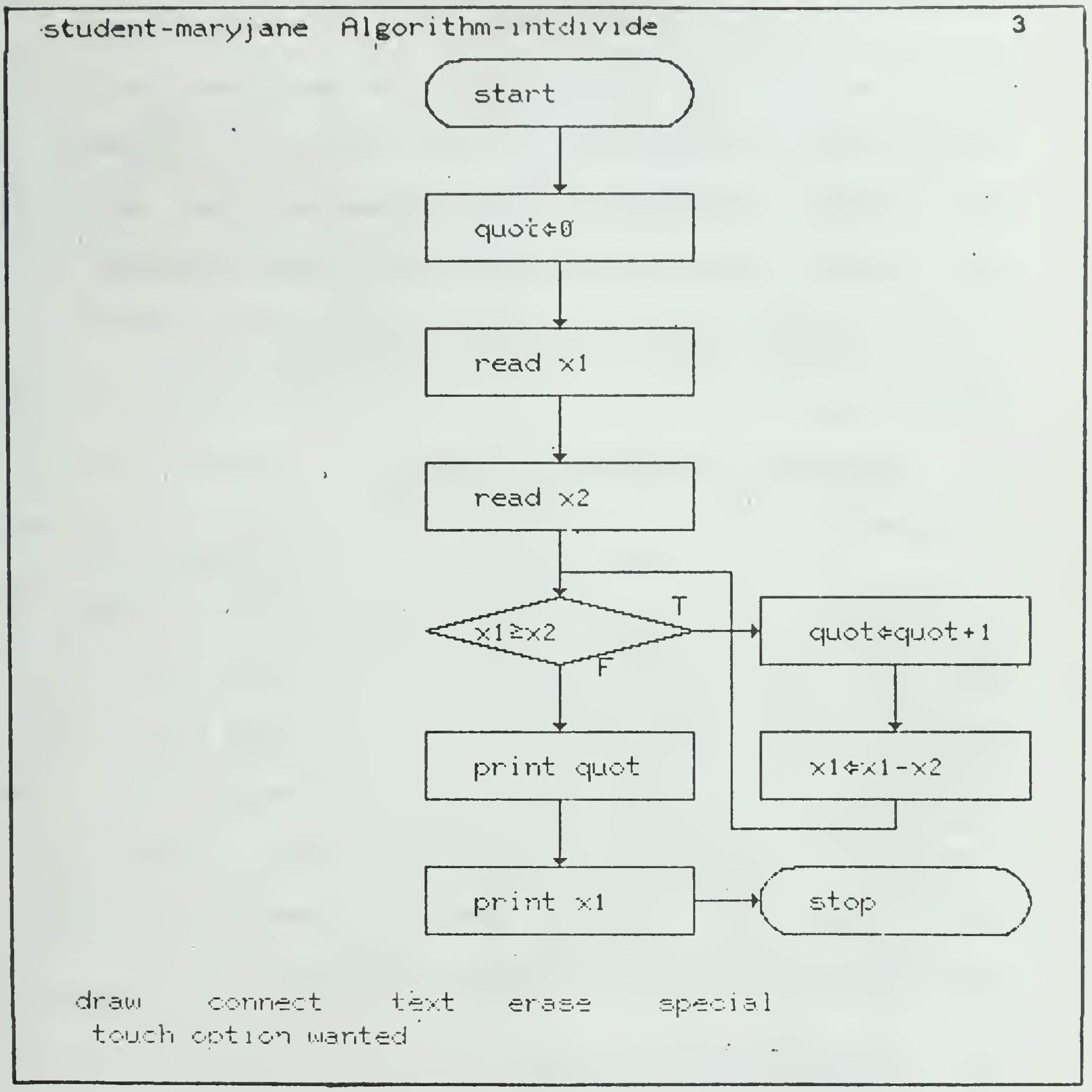


Fig 3-6. Copy of PLATO IV Screen Displaying Maryjane's Algorithm "Intdivide"

3.5.10 Summary of Input Section of PASF

The input section of PASF allows a student, using the touch panel, to "draw" a flowchart of up to twenty-one boxes. Inside the boxes a student may type English phrases or programming code from SIM. The student is given feedback on illegal operations and incorrect syntax as soon as they occur. When the student is ready to have his/her flowchart graded, he/she asks for PASF to check it.

3.6 Checking Student-Generated Flowcharts

Since the students are beginners in programming, the checking part of PASF expects student-generated errors. PASF's goal is to find these errors and report them quickly to the student. A top-down approach searching first for gross errors, then for subtler errors is used. Since, typically, finding gross errors requires less computation, this hierarchical approach burdens the machine less. When an error is found, it is reported to the student, who must then make the necessary correction.

3.6.1 Check for Legal Flowchart

The first check determines whether the student's array of boxes and arrows is a legal flowchart. A legal flowchart consists of one "start," at least one "stop," and all boxes reachable from "start" by traversing the flowchart. A further constraint is that there must be text in every box. The details of the algorithm to check for a legal flowchart, as well as other pertinent algorithms, are covered in Chapter Five. If the flowchart is legal, the student is told that it is a good flowchart, and he/she should press a key to continue checking.

3.6.2 Check for Legal Smooth Flowchart

A basic assumption for PASF is that the student's flowchart is also a smooth flowchart. The next step is an algorithm to check for smoothness. The author's algorithm SR to check for smoothness is given in detail in Chapter Five. A student requires more feedback than the fact that his/her flowchart is not smooth; he/she wants to know where and why. If unsmooth, algorithm SR (cf. Section 5.4) tells the student the offending diamond box, the offending arrow (if the arrow can be determined), and the reason why it is not smooth. As stated before, if a student is following the step-wise refinement procedure, he/she cannot generate a nonsmooth flowchart. In the exercises tried, almost all student flowcharts are smooth. If the flowchart is smooth, a message appears informing the student and tells him/her to press a key to go on.

3.6.3 General Heuristic Checks

The next portion of PASF performs a series of heuristic checks on the student's flowchart. Many students repeatedly make the same types of errors. These heuristic routines attempt to discover common but easily detected errors. For example, undefined variables are common student errors. A simple routine finds variables that the student has forgotten to initiallize or has misspelled. Many heuristic checks are conceptually simple, but nontrivial to execute on a list structure representation. This difficulty initiated a search by the author for an easily manipulated representation of the flowchart.

3.6.4 Regular Expression Representation of Smooth Flowchart

The discovery that a smooth flowchart can be represented as a string allows one to execute many heuristic checks efficiently. Each one of the three constructs of smooth flowcharts shown below can be represented by a string closely akin to regular expressions (RE).

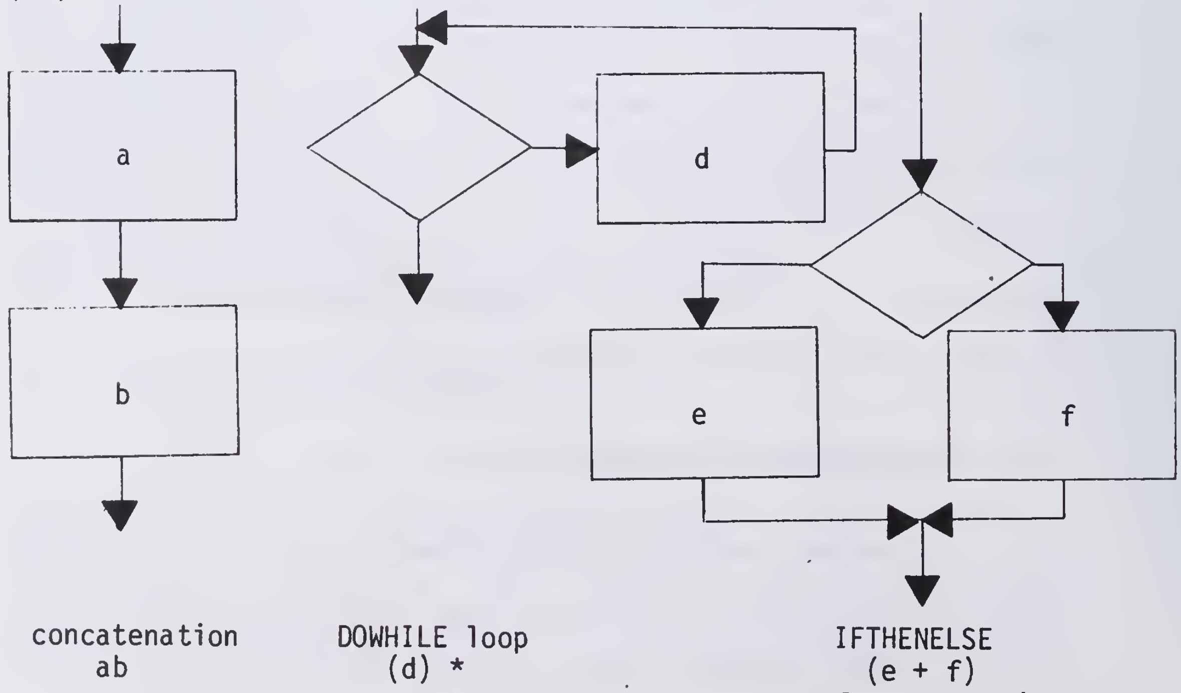
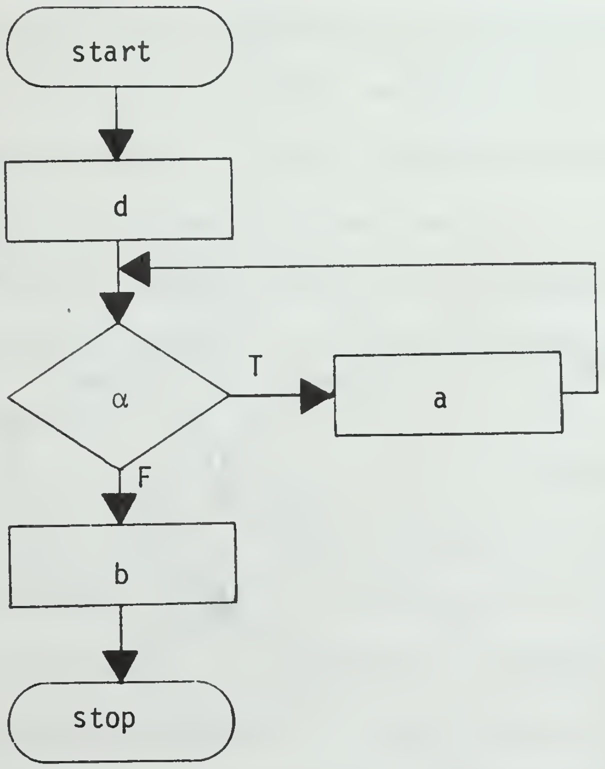


Fig. 3-7. Equivalence of Smooth Flowcharts and Regular Expression Representation

Two rectangles are equivalent to the concatenation of two strings. The DOWHILE, which means to do everything inside the loop zero or more times, is of the form of the Kleene star. The IFTHENELSE is an operator of an "OR" on processes. A few conventions need to be agreed upon before the RE form can completely represent a smooth flowchart. Processes executed first are on the left. A "start" is on the far left of the string and a "stop" is on the far right of the string. Since a smooth flowchart always has one "start" and one "stop," one does not need to write them. From this point, the "start" on the left end and the "stop" on the right end will be understood. Since every diamond corresponds to a



d (a) * [α] b

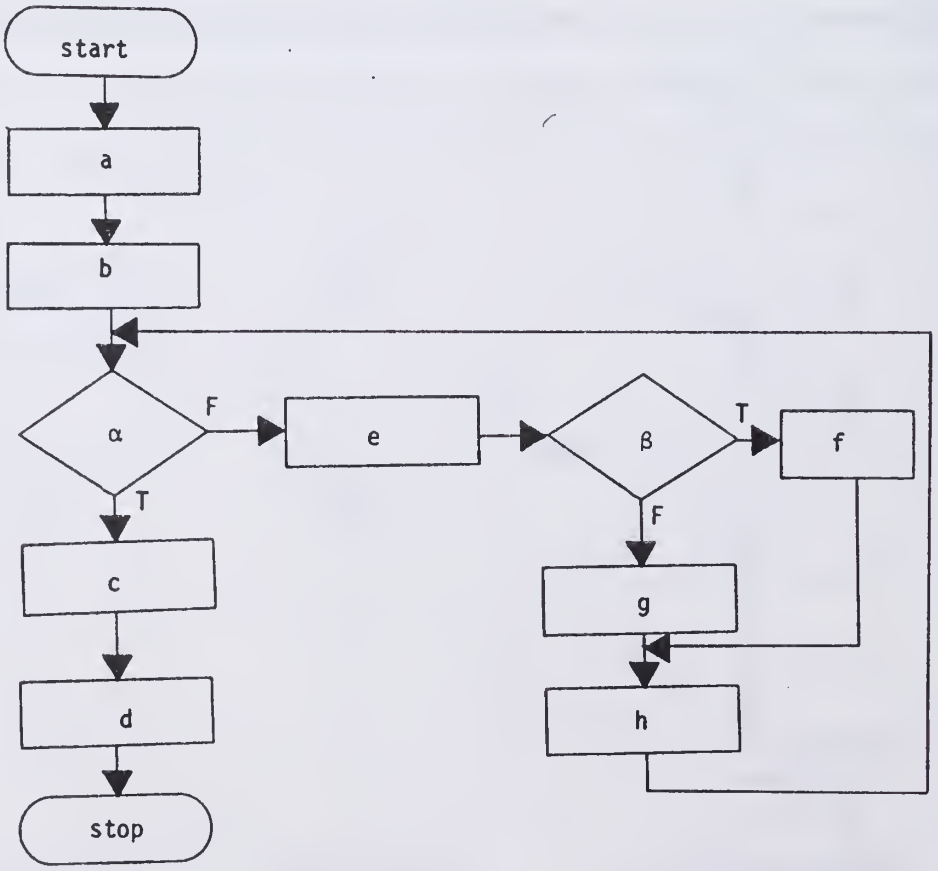
RE Form of C

Smooth flowchart C

Fig. 3-8. RE Form of Smooth Flowchart C

part of a DOWHILE or an IFTHENELSE, the condition inside the diamond will be attached to the corresponding operator "*" or "+" and enclosed in "[]s."

Associated with a diamond box will be a "T" or an "F" to determine which arrow to follow, depending on whether the condition is true or false. For an IFTHENELSE, the true side is defined to be on the left of the "+" in the RE form. For the DOWHILE, the true side is defined to be the branch that loops. If the false side in a flowchart loops, this will necessitate interchanging the "T" and "F" sides by negating the condition.



$ab\ (e(f + [\beta]\ g\ )\ h)\ {*}[\neg\alpha]\ cd$

Fig. 3-9. A Smooth Flowchart D with Its RE Form

Embedded parentheses act in the usual manner as in arithmetic expressions. Below is the RE form of the integer divide smooth flowchart in Fig. 3-6.

```
quot ← 0 read x1 read x2(quot ← quot + 1 x1 ← x1 - x2)*
[x1 ≥ x2] print quot print x1
```

Fig. 3-10. RE Form of Integer Divide Algorithm of Fig. 3-6.

3.6.5 Usefulness of RE Representation of Smooth Flowcharts

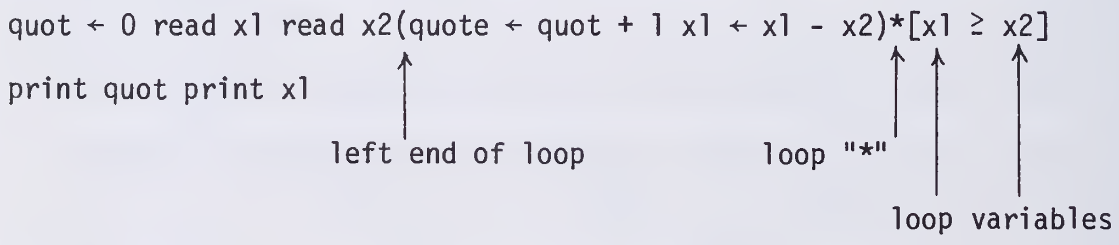
How is this new representation more suited to PASF's needs? In a list structure, one must traverse; in a string, one needs only to scan. Returning to the question of a routine to determine whether any variables are undefined: the RE representation makes this trivial. For each variable var from the symbol table, one needs to scan for "read var" and "var ←." This routine is not only faster to execute than the list structure counterpart, but also uses less storage, is easier to program, and is easier to verify.

As noted in Chapter One, the programming language TUTOR does not have pointers, stacks, recursive procedures, or other list processing features. Any list processing is programmed at a primitive level similar to programming in FORTRAN; any nontrivial routine using list processing is a major effort in programming and verification. On the other hand, TUTOR does have efficient commands to scan strings of characters. These considerations make the RE representation especially suitable.

Before any heuristic checks are performed, the RE representation of the student's smooth flowchart is computed. This computation is done by a slightly modified version of algorithm SR, which checks whether a student's flowchart is smooth or not (cf. Section 5.5). Besides the check to determine whether the variables are undefined, a series of other heuristic checks are done. One more check will be discussed in the next section.

3.6.6 Heuristic Check for Infinite Loops

For all DOWHILE loops, at least one variable in the condition of the diamond must change inside the loop. If this is not true, the flowchart contains an infinite loop. To further illustrate the utility of the RE form, the following process describes this algorithm:

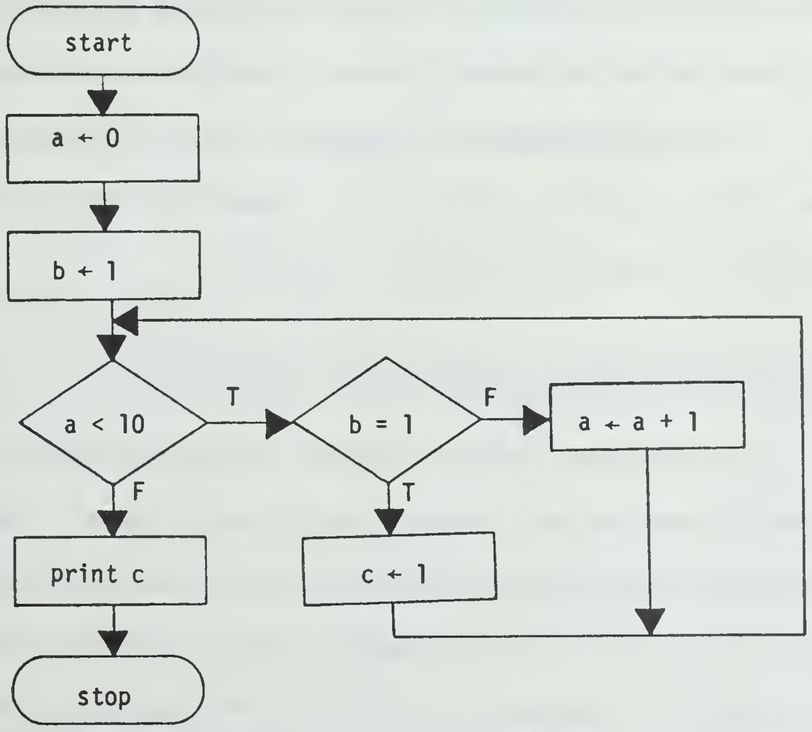


quot ← 0 read x1 read x2(quote ← quot + 1 x1 ← x1 - x2)*[x1 ≥ x2]
print quot print x1

Fig. 3-11. RE Form for Integer Divide to Demonstrate Loop Heuristic, i.e., x1 Changes inside the Loop

First, the program scans for a "*" (used to distinguish loop from "x" which indicates multiplication). Next, the program determines the variables (or loop variables) in the condition by looking a few characters ahead of the "*." The program goes character by character left from the "*" with a pointer, incrementing a counter by one for every ")" and decrementing a counter by one for every "(." The scanner is at the left end of the loop if one encounters a "(" and the count equals zero. The program scans from this "(" to "*," looking for the loop variables to be reassigned. Only one loop variable must be reassigned inside the loop for the heuristic check to pass the test. This is done for all remaining "*"s.

This heuristic check does not guarantee that the variable will actually be changed when the program is executed. In the following example, the heuristic test will pass, but during execution "b" is always "1" and "a ← a + 1" is never done. Therefore, the flowchart in Fig. 3-12 loops forever. The heuristic is unable



a ← 0 b ← 1((c ← 1 + [b = 1] a ← a + 1))* [a < 10] print c

Fig. 3-12. Example in Which Loop Heuristic Passes, but the Flowchart has an Infinite Loop at Execution Time

to detect this case, since it assumes every control (physical) path is executed. Every control path is not an execution path, as is shown in the above example. Testing all execution paths is impractical [Krause, Smith, and Goodwin, 1973].

Furthermore, the set of execution paths for a program will change depending on the data. Run-time dependent analysis of programs is extremely difficult and not attempted in this thesis. The new trend in programming language design is to allow the compiler to do as much optimization, error detection, and verifying as possible. This was a major design criterion in Wirth's computer language PASCAL [Wirth, 1975]. Run-time data dependencies, by definition, can never be done by a compiler. Such dependency checks, e.g., subscript out of range, done at run time are expensive.

After the student's flowchart passes all these heuristic checks, he/she is told, "Fine so far. Now I'll look to see if your flowchart is close to algorithm 'intdivide'."

3.6.7 Algorithm Specific Heuristic Checks of PASF

The previous section of PASF performs heuristic tests suitable for any one of a large class of algorithms. This portion performs algorithm-specific heuristic tests with the necessary information inputted by the instructor. Again, the RE representation of the student's flowchart is used for efficient execution of the tests. The student's flowchart is checked for the correct number of "reads," "prints," DOWHILEs, IFTHENELSEs, English phrases, and other features. If the instructor has indicated that a correct solution should have at least two "read"s, the student is told he/she is missing a "read" (if his/her flowchart contains only one "read"). The student is forced to fix any errors the heuristic tests discover. Finding the number of loops is trivial for PASF: one line of TUTOR code (SEARCH command) is all that is needed to scan the RE representation.

These algorithm-specific heuristics are assurance to PASF before initiating the deduction scheme that the student's attempt at the algorithm is reasonably close to the correct answer. This next and last phase of PASF, the deduction scheme, is expensive in terms of time and computation. Up to this point in the program, PASF's response to a student's input has been fast, taking less than a second of real time. The deduction scheme may take as long as sixty seconds of real time.

3.7 The Deduction Scheme of PASF

The deduction scheme of PASF attempts to show that the student's flowchart is global semantic equivalent (GSE) (cf. Section 2.4.1) to the instructor's flowchart. PASF's goal here is to detect errors and their location as well as to tell the student that his/her flowchart is not correct.

3.7.1 Standard Regular Expression Representation (SRE) of Smooth Flowchart

Before the deduction begins on the two flowcharts, they are transformed into a new representation. The instructor's flowchart, stored in the RE form, is retrieved from read-only-memory (TUTOR Common); the student's flowchart is already in the RE form. This new representation preserves the RE form, but translates each statement, e.g., "a ← a + 1" into a standard syntax. This new representation is called the Standard Regular Expression (SRE) representation. In SRE each statement is of the form:

<type> <sequence number> <ordered list of input variables or constants>:
<ordered list of output variables>;
<ordered list of maybe-output variables>

The possible types of statements in SIM and their new syntax is shown below:

Table 2

Transformation from SIM to Standard Regular Expression (SRE) Form

Statement Type in SIM	SRE Form
a ← 1	int 0 1: a;
b ← c	set 0 c: b;
d ← e + f	add 0 ef: d;
a ← b - c	sub 0 bc: a;
d ← e x f	mul 0 ef: d;
a ← b ÷ c	div 0 bc: a;
print a	out 0 a: ;
read b	inp 0 :b ;
s1	s1 0 : ;
s2	s2 0 : ;
s3	s3 0 : ;
s4	s4 0 : ;

In all the above SRE forms, the sequence number, the use of which will be apparent later, is zero.

The s1, s2, s3, and s4 are codes for distinct English phrases. These codes were determined by the input routine when the student typed in the English phrase.

The order on the list of input variables and constants is the same order as appears in the assignment statement, e.g., "d ← b - c" is transformed to "sub 0 bc: d;", <u>not</u> "sub 0 cb: d;."

The second class of output variables--possible output variables (called maybe-output variables in this thesis)--arises with DOWHILEs and IFTHENELSEs. The "a ← b" inside a loop may never be executed, since that path may not be taken. In this case, "a" is a "maybe-output variable." A similar situation arises with IFTHENELSEs.

In the author's search for a standard format, the rationale for this format is that it mimics a procedure call. Every statement is treated as if it were a procedure call with the input and output variables clearly distinguished. (This is similar to the computer language JOVIAL, which uses ":" to distinguish the input and output variables in a procedure call.)

c ← a + b	call add(a,b,c)	add 0 ab: c ;
Statement	Procedure Call	SRE Form

Fig. 3-13. Procedure Call Analogy of SRE Form

Below is the SRE representation for the integer divide example given in Fig. 3-6.

int 0 0:quot; inp 0 :x1; inp 0 :x2; (add 0 quot 1:quot;sub 0 x1 x2:x1;)* [x1 ≥ x2] out 0 quot:; out 0 x1:;

The above representation may be hard to read for humans (spaces have been added for easier reading in this text), but it is ideal for the machine. The actual representation in the machine contains numbers from 0 to 63 for the different types and pointers as symbol table references.

The deduction scheme scans and manipulates the SRE representations of the two flowcharts.

3.7.2 <u>Block Diagram of Deduction Scheme</u>

Below is a block diagram of the deduction scheme.

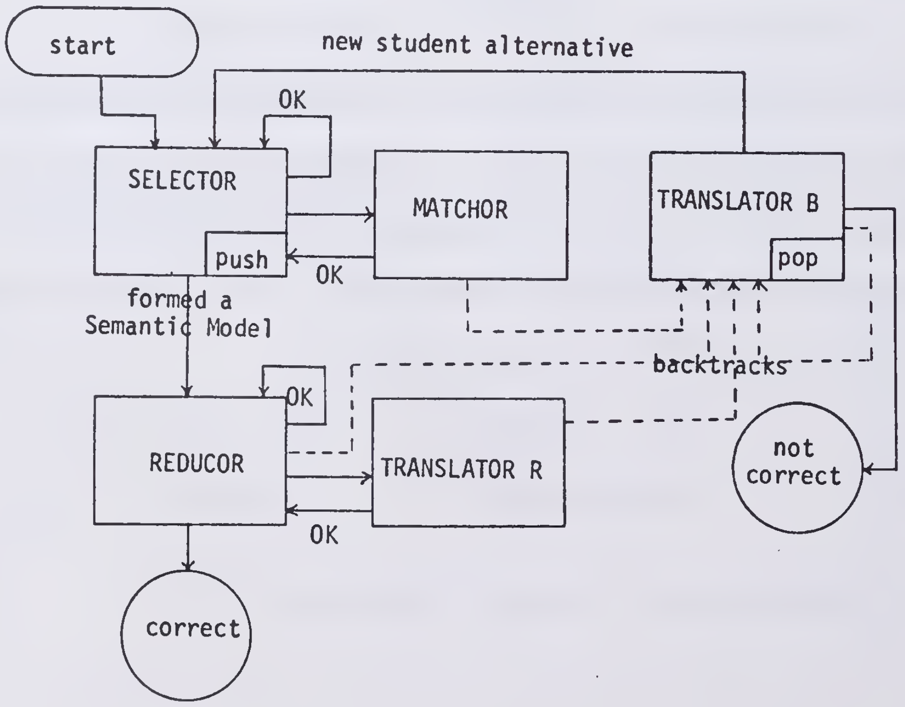


Fig. 3-14. Block Diagram of Deduction Scheme

After the student's and instructor's flowcharts are in SRE form, control of PASF is given to the "SELECTOR." The SELECTOR selects portions of the two flowcharts as candidates for being Global Semantic Equivalent (GSE). The SELECTOR calls the "MATCHOR" to test for a match between the two portions.

When the SELECTOR has produced GSE candidates for all of the two flowcharts, a Semantic Model is generated. Program control is passed to the "REDUCOR" to search the Semantic Model for inconsistencies. If the REDUCOR finds no inconsistencies, the student's flowchart is GSE to the instructor's, i.e., it is correct. In certain cases, the REDUCOR calls the "TRANSLATOR R" to perform translations on the student's flowchart. After such translations, control is returned to the REDUCOR. In cases where the MATCHOR, REDUCOR, or TRANSLATOR R "fail," a backtrack is initiated to generate another Semantic Model. In a backtrack, "TRANSLATOR B" modifies the student's SRE representation before control passes to the SELECTOR.

In the following sections, each portion of PASF will be discussed in detail.

3.7.3 The SELECTOR of Deduction Scheme

It is the task of the SELECTOR, with the help of the MATCHOR, to construct Semantic Models for the REDUCOR. The SELECTOR selects statements from the student's SRE and statements from the instructor's SRE as candidates which may be global semantic equivalent (GSE).[11] Consider the two trivial flowcharts below:

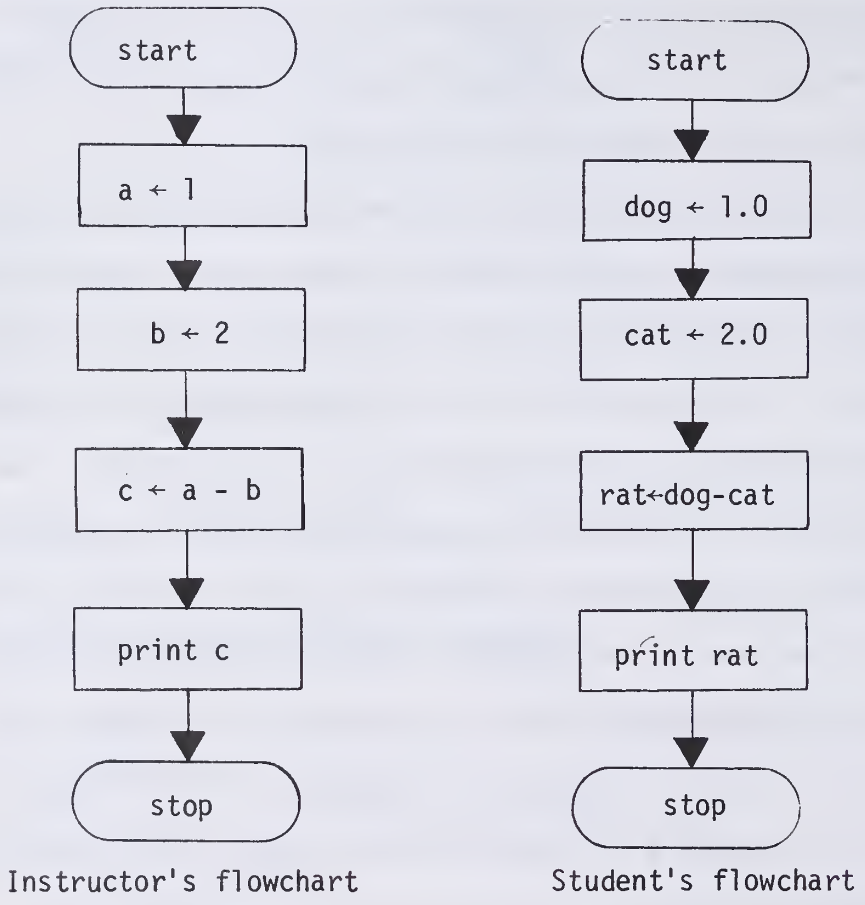


Fig. 3-15. Two Trivial Flowcharts to Demonstrate SELECTOR

First, PASF computes the RE and SRE representations.

Student's RE

dogs ← 1.0   cat ← 2.0   rat ← dog - cat   print rat

Instructor's RE

a ← 1   b ← 2   c ← a - b   print c

---

[11]For discussion of Global Semantic Equivalent, see Section 2.4.1.

```
Student's SRE
int 0 1.0: dog;   int 0 2.0: cat;   sub 0 dog cat: rat;  out 0 rat: ;
Instructor's SRE
int 0 1: a;       int 0 2:b;        sub 0 ab: c;         out 0 c: ;
```

Fig. 3-16. REs and SREs of Flowcharts in Fig. 3-15

The SELECTOR, which is statement-type oriented, scans the student's SRE for "int" and finds the first statement. It searches for and finds an "int" in the instructor's SRE. Finding one in both, the SELECTOR calls the MATCHOR to determine whether the two are locally consistent. In this case the MATCHOR checks to see whether the two constants are equal numerically. Since "1" has the same value as "1.0," the MATCHOR returns an "OK" (cf. Fig. 3-14). Upon receiving the "OK" from the MATCHOR, the SELECTOR assumes that the two "ints" are GSE and constructs part of the Semantic Model by replacing the "ints" with two "P"s (for Process or Program).

```
Student's SRE
P1 1.0: dog;  int 0 2  cat: ; sub 0 dog cat: rat; out 0 rat: ;
Instructor's SRE
P1 1: a;      int 02 : b;     sub 0 a b: c; out 0 c: ;
```

The two P's are given the same sequence number (in this case, "1").

The significance of having any two P's with the same sequence number is that the student's Pi and the instructor's Pi are semantically equivalent, in the programming language sense. They may be GSE, depending on the rest of the SREs. At the moment, they are assumed to be GSE until proven otherwise.

The SELECTOR looks for another "int" and finds an "int" in both SREs. The MATCHOR gives an OK, and the SELECTOR assumes the two are GSE.

Student's SRE

```
P1   1.0: dog;  P2 2.0:cat;  sub 0 dog cat:  rat; out 0 rat: ;
```

Instructor's SRE

```
P1   1.0: a;  P2 2: b;  sub 0 a b: c;  out 0 c: ;
```

The SELECTOR searches for "int" and finds none. Searching through all the statement types, the SELECTOR eventually tries the "out."

Student's SRE

```
P1  1.0: dog; P2 2.0: cat; sub 0 dog cat: rat; P3 rat: ;
```

Instructor's SRE

```
P1  1:  a; P2 2: b; sub 0 a b: c; P3 c: ;
```

The SELECTOR searches for "sub" and calls MATCHOR.

Student's SRE

```
P1 1.0: dog; P2 2.0: cat; P4 dog cat: rat; P3 rat: ;
```

Instructor's SRE

```
P1 1: a; P2 2: b; P4 a b: c; P3 c: ;
```

Since the SELECTOR finds no more statements left with statement-types, it is finished for the time being. The above two SREs constitute the Semantic Model. All the individual pieces of the two flowcharts have been shown to be semantically equivalent. Matching individual pieces does not prove that the pieces fit together properly. Fitting the pieces together is the task of the REDUCOR.

The SELECTOR has other tasks besides generating the Semantic Models. In the process of searching the SRE, the SELECTOR finds all possible alternatives for the candidates to become a P. It pushes the different alternatives along with the current state of the SREs into a stack. Later, TRANSLATOR B will pop this stack in the process of a backtrack. In the example above, the SELECTOR notes the fact that there are two "int"s in each SRE and pushes this fact into the stack.

An important task for the SELECTOR is handling DOWHILE loops and IFTHENELSEs. The SELECTOR finds the first inmost pair of parentheses in both SREs and tries to show that what is inside the parentheses is GSE. In the SREs below, a loop is imbedded inside an IFTHENELSE:

Student's SRE

```
---------- (----- + ----- (    ) * ----- ) -----
                          ↑    ↑
                          sb   se
```

Instructor's SRE

```
---------- ( ----- + ----- (    ) * ----- ) -----
                           ↑    ↑
                           ib   ie
```

The SELECTOR searches for statement-types between the pointers sb and se and the pointers ib and ie; calls the MATCHOR and forms Ps as before; forms a subsemantic model of Ps; and asks the REDUCOR to prove that what is between sb and se is GSE to ib and ie.

If the REDUCOR is successful, one P remains inside each pointer, and the SELECTOR calls the MATCHOR to check whether the conditions in the diamond boxes, e.g., "a ≥ 100," match. If both the REDUCOR and the MATCHOR are satisfied, the SELECTOR reduces the loop of each to a single P. An IFTHENELSE is handled in a similar way. The detailed mechanics of the reduction will be covered later.

In the process of this reduction, the pair of parentheses is eliminated. Now the SELECTOR scans the total SREs to find the next inmost pair of parentheses and continues in like manner until all parentheses are removed from the SREs.

A final task of the SELECTOR is to function as the executive: it is the master that calls the other routines.

3.7.4 The MATCHOR of PASF

The MATCHOR guarantees for the SELECTOR that a piece of the student's flowchart is semantically equivalent to a piece of the instructor's flowchart. If the MATCHOR cannot guarantee this fact, it forces a backtrack and passes control to TRANSLATOR B.

Some pieces of the flowcharts that the MATCHOR handles are single statements. Below are two pieces of program passed by the SELECTOR for the MATCHOR to interrogate:

```
Student      add   0  cat  1.0: dog;
Instructor   add   0  1a: b;
```

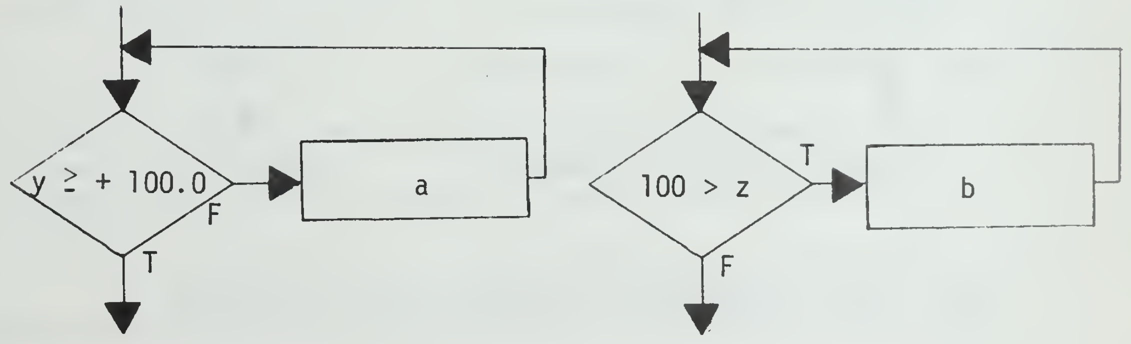
The SELECTOR has verified only that both are "adds"; the MATCHOR must verify that they are both the same form. (Possible forms for an "add" are "add 0 a1 : c;," "add 0 2b : e;," "add 0 12 14: d;," and "add 0 gh : j;," which represent "c ← a + 1," "e ← 2 + b," "d ← 12 + 14," and "j ← g + h" respectively.) In the above example, they are not of the same form, but the MATCHOR notices that the student's can be changed to the same form as the instructor's. The MATCHOR changes the SRE to

Student's add 0 1.0 cat: dog;

since addition is commutative.

Before the above change is made, the MATCHOR checks the values of the constants to see whether they are numerically equal. If the constants are not numerically equal, if the forms are not the same, or if the student's form cannot be changed to the same, the MATCHOR forces a backtrack.

Besides matching statements, the MATCHOR handles the conditions inside diamond boxes. When the SELECTOR finds a loop or an IFTHENELSE, it calls the MATCHOR to match the conditions. The MATCHOR checks the form and the values of any constant. The example below of two flowchart segments demonstrates the task of the MATCHOR.



Student's flowchart segment   Instructor's flowchart segment

Fig. 3-17. Example for MATCHOR

First, all loops are forced to be taken by the "T" side in the RE form. Since the student's flowchart segment has a DOWHILE which loops on "F," the condition inside the diamond is negated.

Student's SRE (a) * [y < +100.0]

Instructor's SRE (b) * [100 > z]

Second, the MATCHOR finds the constants are numerically equal. Third, the MATCHOR knows that "c < d" is semantically equivalent to "d > c." The MATCHOR finds that the two conditions are semantically equivalent and changes the student's SRE to the following to conform to the instructor's:

Student's SRE (a) * [+100.0 > y]

The MATCHOR must also handle English phrases in diamonds.

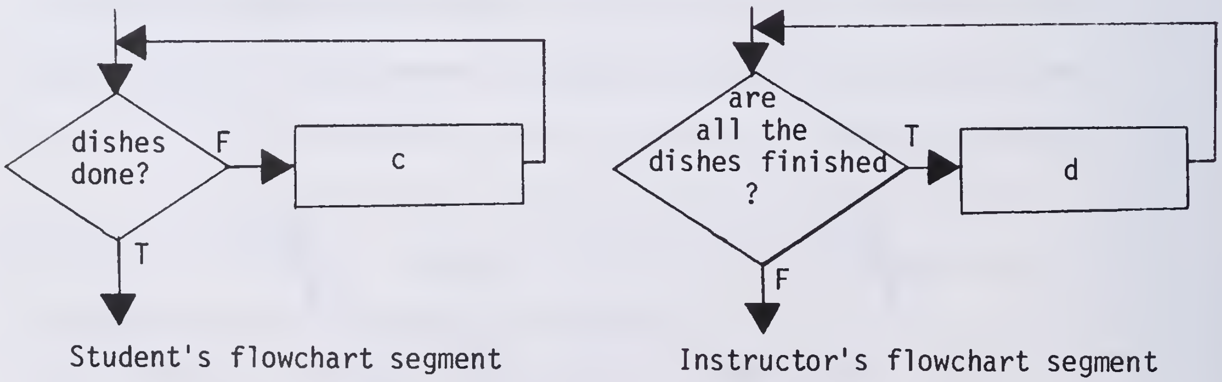

dishes done?
F
c
T
Student's flowchart segment
are all the dishes finished ?
T
d
F
Instructor's flowchart segment

Fig. 3-18. Example with English Phrases in Diamond

The input routine accepts both English phrases and gives them identical names of, for example, "s1." (Of course, this depends on the way the instructor has set up the problem.) When the

RE form is generated, the condition "dishes done?" is negated. The conditions are <u>not</u> semantically equivalent.

Student's SRE Form (c) * [¬ s1]

Instructor's SRE form (c) * [s1]

In this case, the MATCHOR must not only check to see whether i = j of si and sj, but also determine whether they are both negative or not.

When the MATCHOR finds a mismatch in a condition of a diamond box, a message of a possible error is given to the student.

The MATCHOR's task is to try all possible equivalent ways of determining whether two small pieces of a program are semantically equivalent. It may alter the student's SRE slightly if the MATCHOR discovers that the two pieces can then be made semantically equivalent. The MATCHOR's scope of activities is kept to a local level.

3.7.5 The REDUCOR of PASF

The REDUCOR is a crucial part of PASF, since its task is to find an inconsistency in the Semantic Model. The SELECTOR, with the help of the MATCHOR, has generated the Semantic Model. The Semantic Model consists of the student's and the instructor's SRE representations containing all P's. Each individual piece of the student's SRE is semantically equivalent to an individual piece of the instructor's SRE. For the student's total flowchart to be Global Semantic Equivalent (GSE) to the instructor's total flowchart, each individual piece must be GSE as well as semantically equivalent.

During one reduction step, the REDUCOR reduces two <u>adjacent</u> P's in both SREs to one P in both SREs. Assuming the two P's are GSE to the other two P's and certain conditions hold, the REDUCOR forms one P which is assumed to be GSE to the other new P. The diagram below demonstrates one "reduction" step. P3 and P4 have been "reduced" to P5.

$$\begin{tikzcd}
\text{Student's SRE} & P1 & P2 & P4 \arrow[dr] & & P3 \arrow[dl] \\
 & & & & P5 &
\end{tikzcd}$$

(under certain conditions)

$$\begin{tikzcd}
 & & & & P5 & \\
\text{Instructor's SRE} & P1 & P2 & P4 \arrow[ur] & & P3 \arrow[ul]
\end{tikzcd}$$

Student's New SRE   P1 P2 P5

Instructor's New SRE   P1 P2 P5

The goal of PASF's REDUCOR is to "reduce" both SREs, after many steps, to one P. If this can be accomplished, the two flowcharts are GSE and the student is told his/her flowchart is correct.

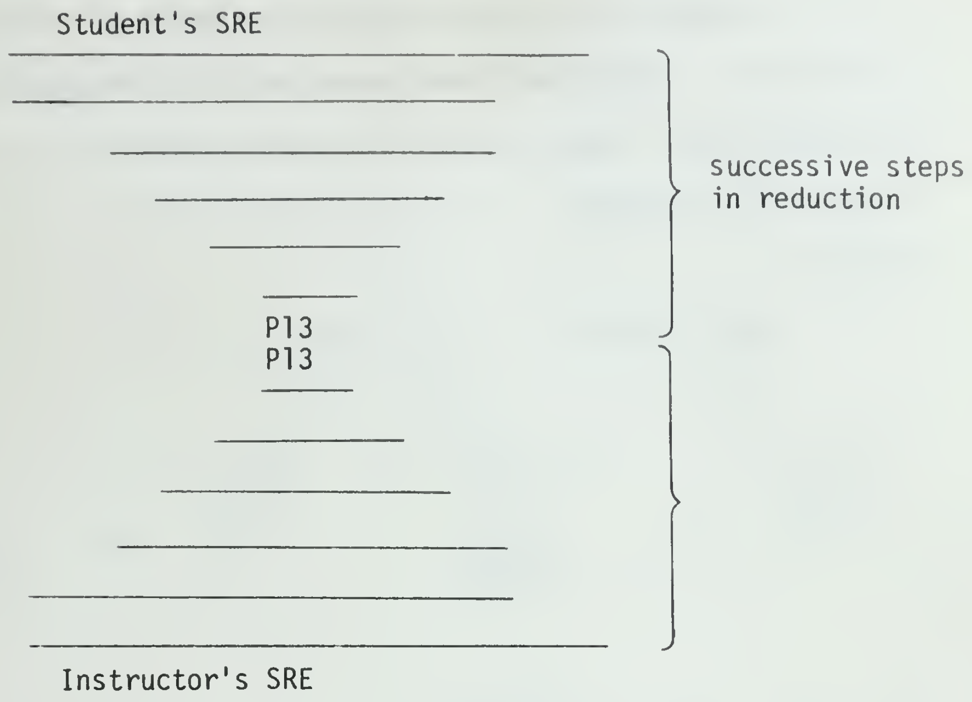
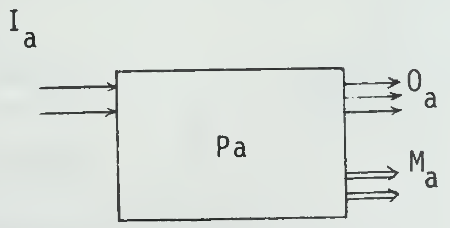


Fig. 3-19. The Goal of REDUCOR: To "Reduce" both SREs to One P

A P can be treated as a black box with inputs and outputs. The inputs, outputs, and maybe-outputs are ordered with the top first.

Pa I_a:O_a;M_a

a "P"



Black Box Representation of a "P"

Fig. 3-20. Comparison of a P and Its Black Box Representation

Reducing two adjacent P's is analogous to combining two black boxes. Seven cases need to be considered in combining the two black boxes. Below is the first case showing how two black boxes can be combined.

CASE 0:

P5 alb:cd;ef   P6 2g:jkl;q

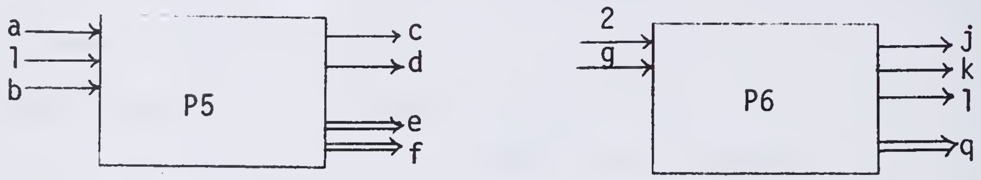


Fig. 3-21. Case 0 Illustrated

Since no input variable, output variable, or maybe-output variable of P5 occurs in P6, the input variables, output variables, and maybe-output variables of the larger P7 are the input variables, output variables, and maybe-output variables of P5 and P6, as shown in Fig. 3-22.

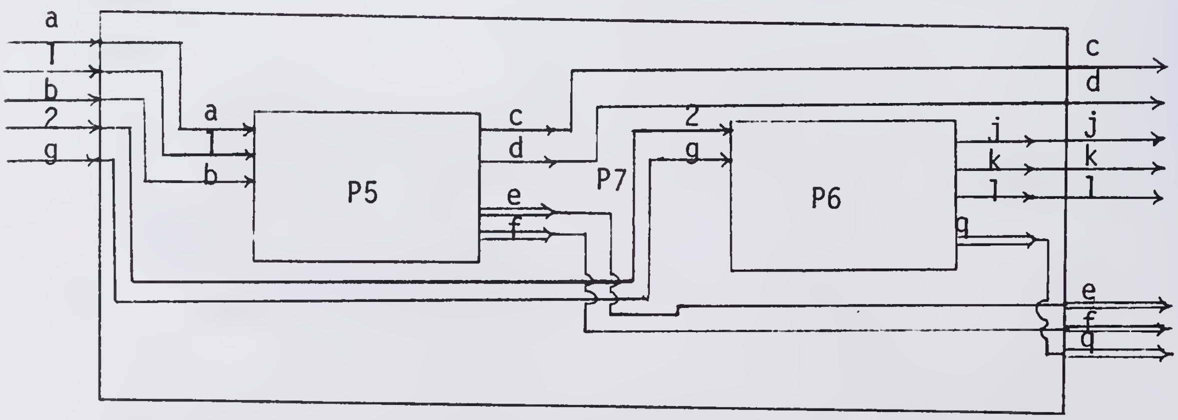


Fig. 3-22. Combined Black Box for Case 0

Case 1 includes an output variable of the first black box which is an input variable of the second black box.

<u>CASE 1:</u>

P1 1.0: dog; P2 dog 2: cat ;

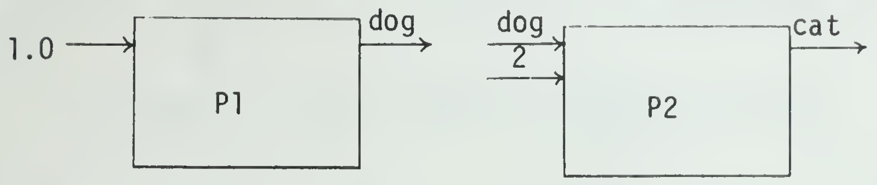


Fig. 3-23. Case 1 Illustrated

The black box on the left (P1) always starts and finishes its computation before the black box on the right (P2) starts.

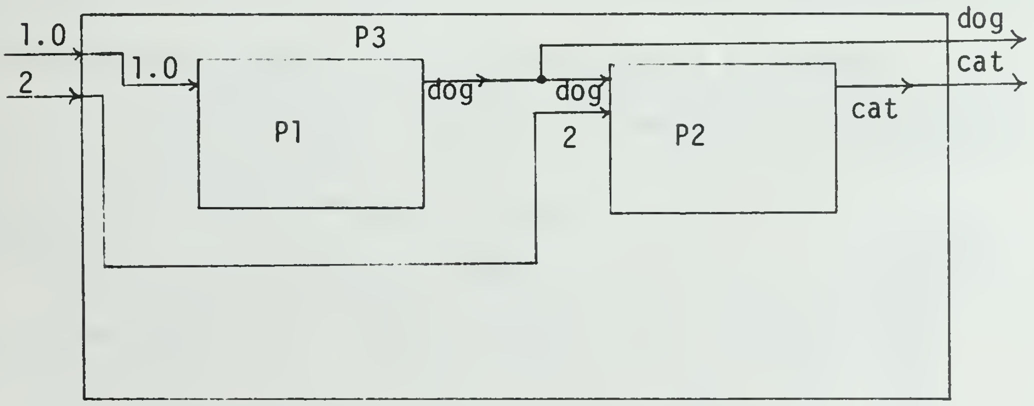


Fig. 3-24. Combined Black Box for Case 1

The two black boxes can be placed into a bigger black box, P3, by giving P3 the proper inputs and outputs. Since "dog" is an output of P1 and an input of P2, it is not an input for the bigger black box P3. The list of inputs of P3 include first the inputs of P1 and then the inputs of P2 (minus "dog"). Similarly, the outputs of P3 are the outputs of P1 and P2, with P1's outputs first. This example shows that an input of the second P can cease to be an input for the bigger box if an output of the first P has the same name.

Case 2 has the same output variable in both P's.

CASE 2:

P1 3:a; P2 4:a; (e.g., $a \leftarrow 3$ $a \leftarrow 4$)

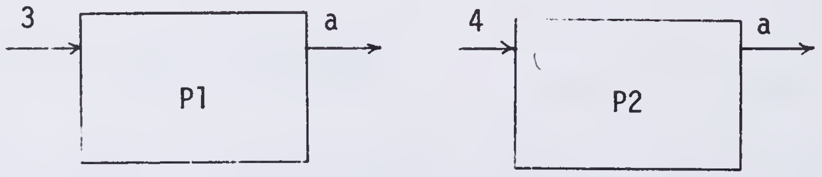


Fig. 3-25. Case 2 Illustrated

Combining these two black boxes reveals that the output variable of P1 is not used as an output variable of P3.

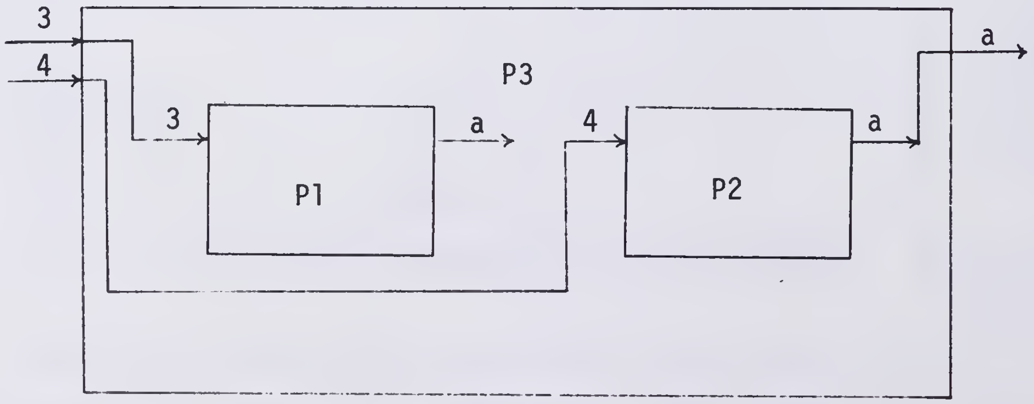


Fig. 3-26. Combined Black Box P3 for Case 2

Because P2 is executed after P1 stops, the value of "a" from P1 is destroyed.

The maybe-output variables arise because of IFTHENELSEs and DOWHILE loops. The following two flowchart segments illustrate the two places where maybe-output variables can occur.

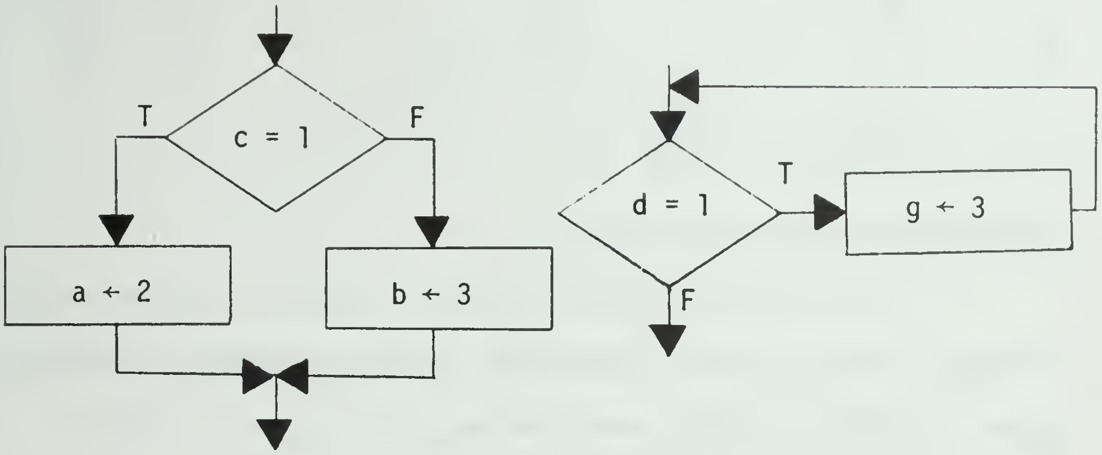


Fig. 3-27. Maybe-Output Variables Arise in Loops and IFTHENELSEs

The variable "a" is a maybe-output variable, for the "T" branch may never be taken. Similarly, "g" is a maybe-output variable, for the loop may never be taken.

The SELECTOR handles the reduction of an IFTHENELSE or a loop if inside there is only two P's or one P, respectively.

$$(\text{Pa } I_a\text{: } O_a\text{; } M_a + [X \text{ op } Y] \text{ Pb } I_b\text{: } O_b\text{; } M_b)$$

is reduced to the following, assuming all conditions are met:

$$\text{Pc XY } I_a I_b\text{: } O_c\text{; } O_a{}'\ O_b{}'\ M_a M_b$$

where 0_c is a list of output variables which appears in both 0_a and 0_b, $0_a'$ is 0_a minus 0_c, and $0_b'$ is 0_b minus 0_c.

Operating similarly for loops, the SELECTOR reduces the following

$$(Pa\ I_a\colon 0_a;\ M_a) \quad * \quad [X\ op\ Y]$$

to

$$PcXY\ I_a\colon ;\ 0_a\ M_a$$

assuming all conditions are met.

In both the loop and IFTHENELSE reductions, maybe-output variables are generated. Discussion of the occurrences of maybe-output variables in black boxes follows (Cases 3, 4, 5, and 6). The flowchart-segment below illustrates Case 3.

<u>CASE 3:</u>

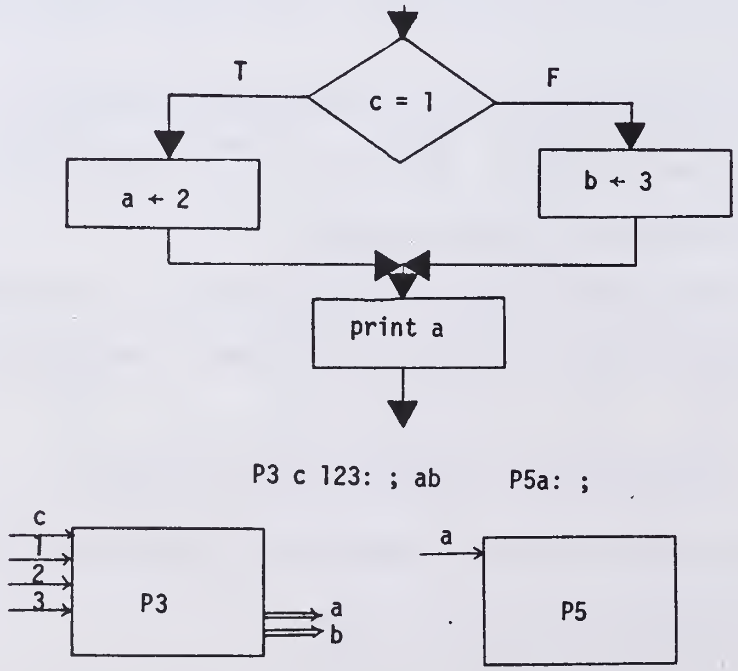


Fig. 3-28. Flowchart Segment Illustrating Case 3

Combining the two black boxes, one must decide what to do with the maybe-output variable "a."

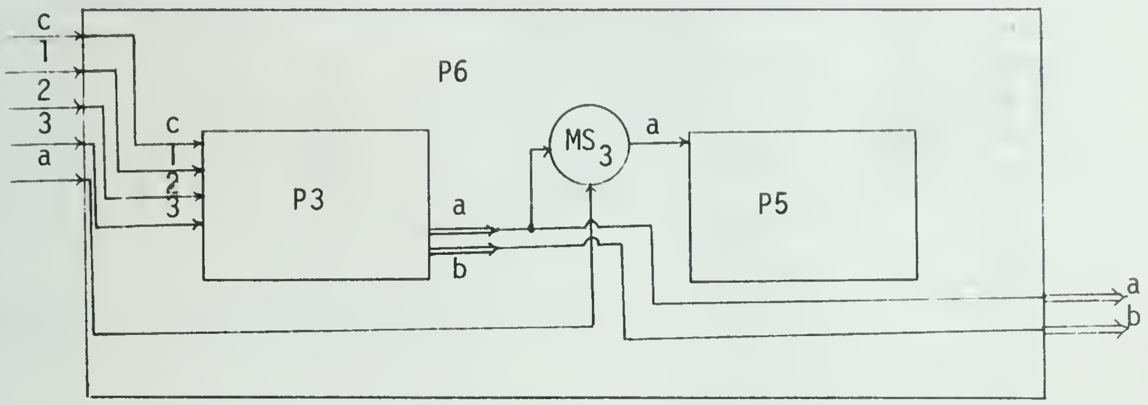


Fig. 3-29. Combined Black Box P6 for Case 3

A Maybe-Switch (MS) is incorporated into the larger black box, P6, above. If the variable "a" inside P3 is an output variable, the MS switches the input line of P5 to output of P3; otherwise, the MS switches the input line of P5 to the input line of P6.

The two P's below with "a ← 8" replacing the previous "print a" constitute an example of Case 4.

P3 c123: ; ab P6 8: a ;

<u>CASE 4:</u>

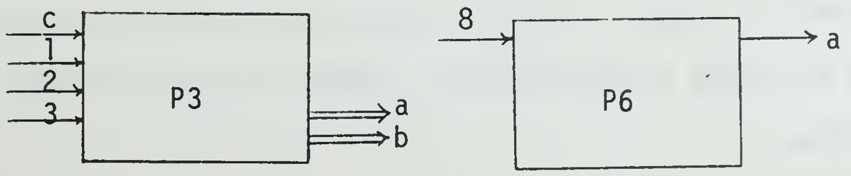


Fig. 3-30. Case 4 Illustrated

When the two black boxes are combined, the "a" of P6 "destroys" the "a" of P3.

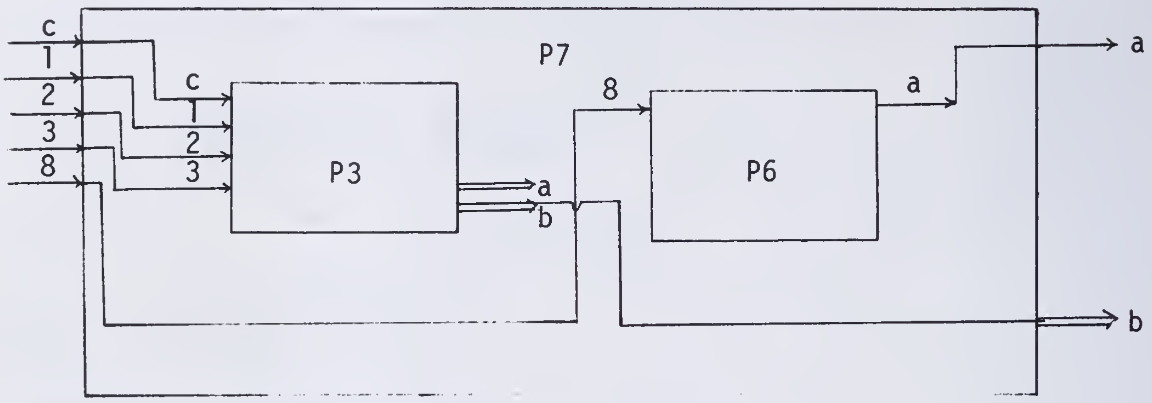


Fig. 3-31. Combined Black Box P7 for Case 4

Reversing the two P's shown in Case 4 illustrates an example of Case 5.

P6 8:a;   P3 c123:;ab

CASE 5:

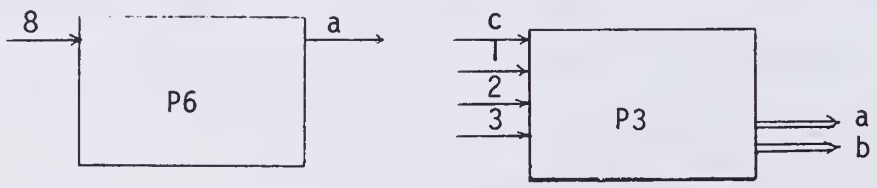


Fig. 3-32. Case 5 Illustrated

The fact that a variable is an output variable in either small black box makes that variable an output variable for the larger black box.

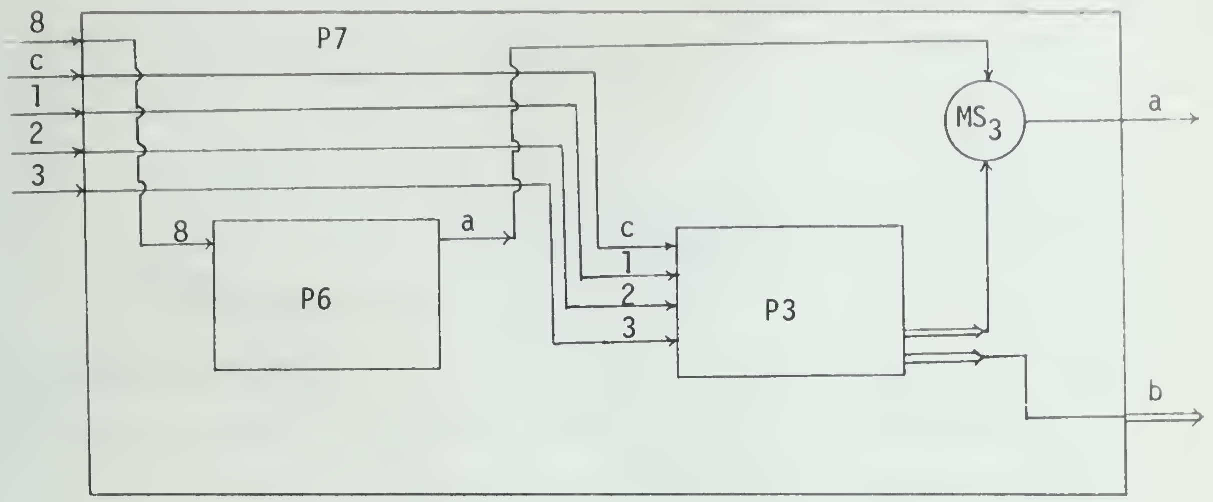


Fig. 3-33. Combined Black Box for Case 5

In the last case, the two P's below with "a" are a maybe-output variable in both P's (replacing the "a ← 8" by another IFTHENELSE).

CASE 6:

P3 c123: ; ab   P7 c245:; ad

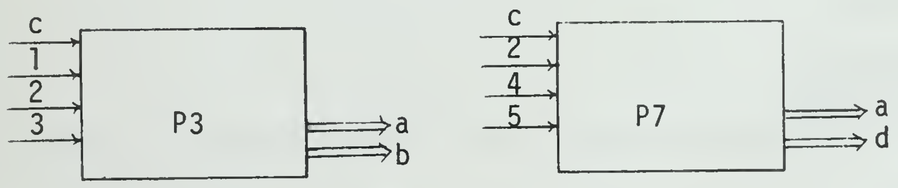


Fig. 3-34. Case 6 Illustrated

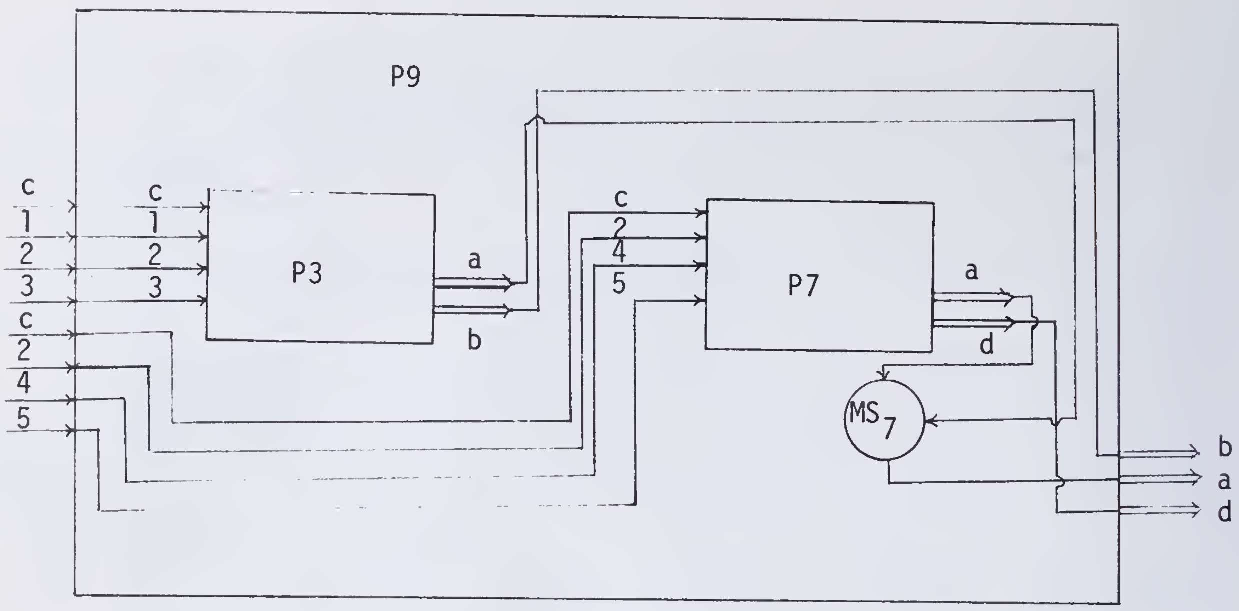


Fig. 3-35. Combined Black Box P9 for Case 6

An MS is incorporated to "destroy" the "a" of P3 if the "a" in P7 is an output variable.

Seven cases have been considered in combining two smaller black boxes into a larger one. These seven cases will be used in the development of the rules for reducing two P's of the instructor's SRE with two P's of the student's SRE.

Student's SRE	Pa I_a: O_a; M_a	Pb I_b: O_b; M_b
Instructor's SRE	Pc I_c: O_c; M_c	Pd I_d: O_d; M_d

In the above two SREs if the sequence number "a" equals the sequence number "b," the two P's (Pa and Pc) are semantically equivalent; similarly, Pb and Pd are semantically equivalent if the sequence numbers b and d are equal. If Pc and Pd are adjacent, they may be combined by the black box technique; if Pa and Pb are adjacent, they likewise may be combined by the black box technique. If all these conditions exist, then an attempt can be made to "reduce" Pa and Pb, and Pc and Pd. These conditions are stated as Rules 1 and 2.

<u>Rule 1 for Reduction</u>:

The first P of each set of P's to be reduced must have the same sequence number, e.g., a = c.

<u>Rule 2 for Reduction</u>:

The second P in the student's set of P's to be reduced must have the same sequence number as the second P in the instructor's SRE.

If Rules 1 and 2 are satisfied, then two sets of two black boxes can be drawn. Rules 3 through 14 refer to Fig. 3-36.

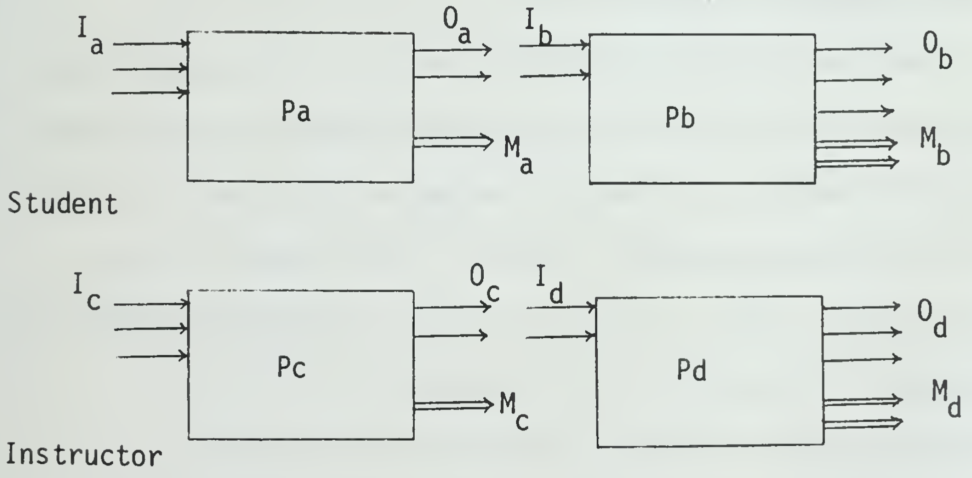


Fig. 3-36. The Set of P's to be Reduced

Not only are Pa and Pc semantically equivalent, but I_a and I_c have the same number of inputs. Any constants in I_c appear in the same position in I_a, and the constants have the same numerical value. The number of output variables in O_a equals the number of output variables in O_c; a similar situation exists with M_a and M_c. Furthermore, the SELECTOR has guaranteed that these facts apply to the inputs and outputs of not only Pa and Pc, but also of Pb and Pd. The corresponding pieces, e.g., inputs, outputs, etc., before being combined have the same structure.

When Pc and Pd are combined to Pf, the seven cases discussed previously will be applied to form the inputs, outputs, and maybe-outputs. If, when Pa and Pb are combined to Pe the same internal structure and corresponding inputs, outputs, and maybe-outputs occur as in Pf, then Pe and Pf can be formed. A "reduction" takes place when Pc and Pd are combined to form Pf, and Pa and Pb are combined to form Pe.

The next two rules result from Case 1, which states that an output variable of the first black box is internally "connected" to the same variable as an input to the second black box. Rule 3 checks for missing occurrences of Case 1 in the student's set of P's to be reduced. Rule 4 checks for extra occurrences of Case 1 in the student's set of P's to be reduced.

<u>Rule 3 for Reduction</u>:

> Any output variable in O_c that occurs in I_d must also have a corresponding (by position) output variable in O_a which is repeated in I_b; further, the position of the input variable in I_b must be equal to the position of the input variable in I_d.
>
> If $O_{cj} = I_{dk}$ then $O_{aj} = I_{bk}$

Rule 4 for Reduction:

There exist no output variables in O_a occurring in I_b and repeated as a corresponding output variable in O_c which are not equal to a variable in I_d.

$$\neg\exists \quad O_{aj} = I_{bk} \text{ such that } O_{cj} \neq I_{dk}$$

The following two rules result from Case 2, where an output variable of the second black box can "destroy" the same output variable of the first black box.

Rule 5 for Reduction:

Any output variable in O_c which is repeated in O_d must have a corresponding output variable in O_a which is repeated in O_b. The position of the output variables in O_d and O_b must be the same.

$$\text{If} \quad O_{cj} = O_{dk} \quad \text{then} \quad O_{aj} = O_{bk}$$

Rule 6 for Reduction:

There exist no output variables in O_a occurring in O_b and repeated as a corresponding output variable in O_c which are not equal to a variable in O_d.

$$\neg\exists \quad O_{aj} = O_{bk} \text{ such that } O_{cj} \neq O_{dk}$$

In considering Case 3, any Maybe-Switch is related to whether the maybe-output variable is actually changed or not inside the first P. A maybe-output variable is changed if a proper path through the loops and IFTHENELSEs is taken. If Pa and Pc are semantically equivalent, then any path taken in Pc will have a corresponding path taken in Pa; therefore, a Maybe-Switch related to Pc will be set identically to a corresponding Maybe-Switch related to Pa.

The actual setting of a Maybe-Switch is unimportant; what is important is that if Pe has a Maybe-Switch for variable "a," then Pf must have a Maybe-Switch for the corresponding variable. The check for the occurrence of a Maybe-Switch is done by Rules 7 and 8, which are quite similar to Rules 3 and 4.

Rule 7 for Reduction:

Any maybe-output variable in M_c that is repeated in I_d must also have a corresponding (by position) maybe-output variable in M_a which is repeated in I_b. Further, the position of the input variable in I_b must equal the position of the input variable in I_d.

If $M_{cj} = I_{dk}$ then $M_{aj} = I_{bk}$

Rule 8 for Reduction:

There exist no maybe-output variables in M_a occurring in I_b and repeated as corresponding output variables in M_c which are not equal to a variable in I_d.

$\neg\exists\ M_{aj} = I_{bk}$ such that $M_{cj} \neq I_{dk}$

Rules 9 and 10 for Case 4 are similar to Rules 5 and 6. The latter pair deal with output variables; the former pair deal with maybe-output variables.

Rule 9 for Reduction:

Any output variable in O_d which is the same as a maybe-output variable in M_c must have a corresponding output variable in O_b which is the same as a maybe-output variable in M_a. The position of the maybe-output variables in M_c and M_a must be equal.

If $O_{dj} = M_{ck}$ then $O_{bj} = M_{ak}$

Rule 10 for Reduction:

There exists no output variable in O_b occurring as maybe-output variable in M_a and repeated as corresponding output variable in O_d which is not equal to a maybe-output variable in M_c.

$\neg \exists \quad O_{bj} = M_{ak}$ such that $O_{dj} \neq M_{ck}$

For Case 5, a Maybe-Switch is incorporated which switches the output variable of the two smaller black boxes to the output variable of the larger black box. Rules 11 and 12 check for a corresponding Maybe-Switch in both P's.

Rule 11 for Reduction:

Any output variable in O_c which is repeated as a maybe-output variable in M_d must have a corresponding output variable in O_a which is repeated as a maybe-output variable in M_b. The position of the maybe-output variables in M_d and M_b must be the same.

If $O_{cj} = M_{dk}$ then $O_{aj} = M_{bk}$

Rule 12 for Reduction:

There exist no output variables in O_a occurring in M_b and repeated as a corresponding output variable in O_c which is not equal to the maybe-output variable in M_d.

$$\neg \exists \quad O_{aj} = M_{bk} \text{ such that } O_{cj} \neq M_{dk}$$

For Case 6 the same argument about the Maybe-Switch holds true. If Pb and Pd are semantically equivalent, any Maybe-Switch related to Pb will be set identically to a corresponding Maybe-Switch related to Pd. Rules 13 and 14 are similar to Rules 5 and 6.

Rule 13 for Reduction:

Any maybe-output variable in M_d which is repeated as a maybe-output variable in M_c must have corresponding maybe-output variable in M_b which is repeated as a maybe-output variable in M_a. The position of the maybe-output variables in M_c and M_a must be the same.

$$\text{If } M_{dj} = M_{ck} \text{ then } M_{bj} = M_{ak}$$

Rule 14 for Reduction:

There exist no maybe-output variables in M_b occurring in M_a and repeated as a corresponding maybe-output variable in M_d which is not equal to the maybe-output variable in M_c.

$$\neg \exists \quad M_{bj} = M_{ak} \text{ such that } M_{dj} \neq M_{ck}$$

Rule 14 completes the set of rules which must be satisfied in order to do a reduction.

The Rules 7, 8, 13, and 14 are stronger conditions than are required. In the situations when the variables are not changed, these rules need not be applied, but discovering such situations is very time-consuming or impossible (closely akin to finding all traces in a program). In many situations, the maybe-output variables are changed depending on run-time data; therefore, to cover the important situations even when it is not known which are the important ones, the author has decided to apply Rules 7, 8, 13, and 14 all the time.

Inspecting the fourteen rules, one sees that some, e.g., Rules 3 and 7, resemble each other very much. This fact is used in the actual implementation of PASF.

The fourteen rules above are tests which are applied to the set of P's to be reduced <u>before</u> the reduction. If the set of P's to be reduced passes all fourteen tests, then the actual reduction takes place as below,

Student's SRE	Pa I_a: O_a; M_a	Pb I_b: O_b; M_b
Instructor's SRE	Pc I_c: O_c; M_c	Pd I_d: O_d; M_d
Student's New SRE after Reduction	Pe I_a I_b':O_a' O_b; M_a' M_b'	
Instructor's New SRE after Reduction	Pf I_c I_d': O_c' O_d; M_c' M_d'	

Fig. 3-37. A Reduction Step

where e = f (a new sequence number), I_b' and I_d' are I_b and I_d minus the input variables mentioned in Rule 3, O_a' and O_c' are O_a and O_c minus the output variables mentioned in Rule 5. M_a' and M_c' are M_a and M_c minus the maybe-output variables mentioned in Rule 9 and M_b' and M_d' are M_b and M_d minus the maybe-output variables in Rule 11.

These fourteen rules look like a lot of work, but, in practice, they are easy to apply and require only a modest amount of code; furthermore, many times the tests are trivial, since the P's may have neither maybe-output variables nor even output variables.

The following example demonstrates the application of the fourteen rules and the reduction process. In Section 3.7.3, the SELECTOR generated the following Semantic Model as an example (cf. Fig. 3-16):

Student's SRE

```
P1  1.0: dog; P2 2.0: cat; P4 dog cat: rat; P3 rat: ;
                           ↑                ↑
                           Pa               Pb
```

Instructor's SRE

```
P1  1: a;  P2  2:b;  P4 ab: c; P3 c:;
                     ↑         ↑
                     Pc        Pd
```

The REDUCOR first tries to reduce the P with the highest sequence number--in this case, P4. Rule 1 is applied to find P4 in the student's SRE. Rule 2 is satisfied, since the P immediately following P4 in the student's SRE has the same number (P3) as the P immediately following P4 in the instructor's SRE.

Applying Rule 3, the REDUCOR finds that the output variable "c" of the instructor's P4 is an input variable in P3 of the instructor's. For Rule 3 to be satisfied, the first output variable of P4 of the student's must occur as the first input variable in P3 of the student's. In this case, "rat" is in the proper positions. The application of Rules 4 through 14 reveals no complications or difficulties.

When all the fourteen rules have been satisfied, P3 and P4 of both can be reduced. First, "c" in P3 and "rat" in P4 are eliminated, a new sequence number not used elsewhere is given to the new P, and the input and output variables are combined as shown previously (cf. Fig. 3-37).

```
Student's New SRE
after One Reduction    P1  1.0: dog;  P2 2.0: cat;  P5 dog cat: rat;

Instructor's New SRE
after One Reduction    P1  1:a;  P2 2: b;  P5  ab: c;
```

Taking the SREs another reduction step, the REDUCOR searches for the highest number P (P5) and tries to reduce it. Since there is no P following P5, the REDUCOR searches for the next lowest numbered P (P2). Since P5 follows P2 in both the instructor's and the student's SREs, Rules 1 and 2 are satisfied. Rule 3 eliminates the "b" in P5 and the "cat" in P5. Rules 4 through 14 are applied, but do not affect the P's.

```
Student's SRE
after Two Reductions   P1  1.0 : dog;   P6  2.0 dog : cat rat;

Instructor's SRE
after Two Reductions   P1  1: a;  P6  2a:  bc ;
```

After the third reduction, only one P is left in each of the SREs.

```
Student's SRE
after Three Reductions   P7  1.0  2.0 : dog cat rat ;

Instructor's SRE
after Three Reductions   P7  1  2  :  abc ;
```

If the student's SRE and the instructor's SRE are both reduced to one P, then the two flowcharts are GSE. The student is told his/her flowchart is correct.

The fact that only constants are left as inputs in P7 is always true for this method. The completely reduced SRE of one P will always list all the constants and all the variables used in the program.

If the test corresponding to Rule 2 fails, the REDUCOR calls TRANSLATOR R. If the test for Rules 3 through 14 fail, the REDUCOR stops and forces a backtrack.

3.7.6 The TRANSLATOR R of PASF

If Rule 2 of the reduction fails, the REDUCOR calls TRANSLATOR R. TRANSLATOR R tries to move around the P's in the student's SRE to satisfy Rule 2.

Rule 2 demands that the second P after Pi (the first P trying to be reduced) in the student's SRE have the same sequence number as the second P after Pi in the instructor's SRE. The second P will be called Pj. There are two possible cases for the Pj in the student's SRE in relation to Pi. The first case is to have other P's (Pc•••Pz) between Pi and Pj.

```
Student's SRE       ------------ Pi Pc•••Pz Pj -------
Instructor's SRE    ---------Pi   Pj ------------
```

There are two ways to move the student's Pi and Pj together. The first is to interchange "Pc•••Pz" and "Pj"; the second is to interchange "Pi" and "Pc•••Pz."[12] The first is tried; if it cannot be done, the second is tried. If either succeeds, TRANSLATOR R returns control to the REDUCOR to continue the reduction; if neither succeeds, a backtrack is forced.

The second case is to have Pj before Pi, possibly with several P's in between.

```
Student's SRE       ----------  Pj  Pc•••Pz  Pi ---------
Instructor's SRE    ----------  Pi  Pj ------------
```

Again, there are two ways to move the student's Pi Pj together: the first is to interchange "Pc•••Pz Pi" and "Pj"; the second is to interchange "PjPc•••Pz" and "Pi." If either succeeds, TRANSLATOR R returns control to the REDUCOR; if both fail, then a backtrack is forced. The conditions necessary in order to interchange P's are covered in the next section.

3.7.7 <u>Conditions Which Allow Interchanging of Portions of Student's SRE</u>

Considering A and B as arbitrary statement types (e.g., add Oab:c;), as arbitrary P's, or as arbitrary strings of P's and

[12]There are special conditions which must be satisfied in order to perform such an interchange, but they will be covered in Section 3.7.7.

statement types, AB can be interchanged to form BA if the following three conditions are true:

Condition 1

If no output variable or maybe-output variable of A is an input variable of B.

Condition 2

If no output variable or maybe-output variable of B is an input variable of A.

Condition 3

If no output variable or maybe-output variable of A is an output variable or maybe-output variable of B.

Only these three conditions need be satisfied in order to move two strings of SREs around.

Below are a few examples to demonstrate the three conditions.

Example 1

The following cannot be interchanged, because "a" is an output variable of P1 and an input variable of P2. (Condition 1 fails.)

```
        a ← 1   b ← a + 2
SRE     P1  1: a;  P2 a2: b;
```

Example 2

The following cannot be interchanged. (Condition 3 fails.)

```
        a ← 12   a ← 4
SRE     P1  12: a ;    P2  4: a ;
```

Example 3

(All three conditions pass.)

read a   b $\leftarrow$ 0   i $\leftarrow$ 0   read d   print d

with X and Y as shown

SRE with some P's $\underbrace{\text{inp 0 : a;   P1 0: b;   P2 0: 1;}}_{X}$ $\underbrace{\text{inp 0 : d; out 0 d: ;}}_{Y}$

This interchange is possible since X and Y satisfy all three conditions.

SRE after inter-change $\underbrace{\text{inp 0 : d; out 0 d: ;}}_{Y}$ $\underbrace{\text{inp 0 : a ; P1 0 : b; P2 0 : 1;}}_{X}$

3.7.8 The TRANSLATOR B of PASF

Any backtrack request is handled by TRANSLATOR B (cf. Fig. 3-14). If the MATCHOR, REDUCOR, or TRANSLATOR R "fail" in their tasks, they force a backtrack and pass control to TRANSLATOR B. TRANSLATOR B pops the stack, manipulates the student's SRE to try the next possible alternative, and passes control to the SELECTOR, which generates a new Semantic Model for the REDUCOR to check. In an attempt to pop an empty stack, TRANSLATOR B halts the deduction scheme and outputs the message to the student that his/her flowchart is either not correct or not close enough to correct.

In the process of generating the Semantic Model, the SELECTOR searches for all possible alternatives of GSE candidates. When the SELECTOR finds an alternative, it pushes into the stack the current state of the student's SRE, the current state of the instructor's SRE, a few pointers, and the reason why there is an alternative. When TRANSLATOR B pops the stack, enough information is in the stack to restart the SELECTOR at the new state. This new state, which is a partial solution, may have a mixture of statement types, e.g., "add 0 1 a: a;," and "P's." In fact, the partial solution may be close to being finished, e.g., with a DOWHILE loop reduced to a single P. TRANSLATOR B, before passing control to the SELECTOR, must do two things.

First, TRANSLATOR B makes a modification to the student's SRE which corresponds to the new alternative. For example, a simple case arises when the SELECTOR finds an "add" of the form "add 0 b c: d;." An alternative is "add 0 c b: d;," since addition is commutative. This fact is pushed into the stack. After popping the stack, TRANSLATOR B alters the student's "add" and the deduction scheme tries this new SRE.

The second task of the TRANSLATOR B is to set a flag used by the SELECTOR or MATCHOR to guarantee that the deduction scheme does not attempt an alternative tried previously. Without this feature, the deduction scheme loops forever, pushing and popping on the same alternative.

A common alternative pushed by the SELECTOR is multiple statement types. The example below contains multiple "int s" ("a ← 1" form).

```
Student's SRE      int 0 1: zap;   P1 P2   int 0 1: cow; --------
Instructor's SRE   int 0 1: a; P2 P1  int 0 1: b; ----------
```

There is no way to tell whether "int 0 1: zap;" is GSE to "int 0 1: a;" or to "int 0 1: b;." Both ways must be tried. The original way is tried first, and then the second way is pushed into the stack. After popping the stack, TRANSLATOR B tries to interchange "int 0 1: zap;" with "int 0 1: cow;." In general, there will be statement types or P's in between. Section 3.7.7 gives three conditions necessary to interchange AB to BA, but in this case XZY to YZX is needed. The three rules for interchanging AB to Ba can be used for XZY to YZX, but they must be applied twice.

First Application of Interchange Rules $\quad \underset{\overset{\uparrow}{A}}{X}\,\underset{\overset{\uparrow}{B}}{\underbrace{ZY}} \qquad \underset{\overset{\uparrow}{B}}{\underbrace{ZY}}\,\underset{\overset{\uparrow}{A}}{X}$

Second Application of Interchange Rules $\quad \underset{\overset{\nearrow}{A'}}{Z}\,\underset{\overset{\nwarrow}{B'}}{Y}\,X \qquad \underset{\overset{\nearrow}{B'}}{Y}\,\underset{\overset{\nwarrow}{A'}}{Z}\,X$

TRANSLATOR B checks these six conditions before attempting the move. The checks may fail for several reasons: Z and Y may be dependent and cannot be interchanged, X and Z may be dependent and cannot be interchanged, or other reasons. If Z and Y are dependent, the TRANSLATOR B tries to move XZY to ZYX. (Placing the X after the Y forms a different alternative as the SELECTOR scans from left to right.) If X and Z are dependent, the TRANSLATOR B tries to move XZY to YXZ. If all three attempts to interchange fail, TRANSLATOR B forces a backtrack and calls itself to pop another item from the stack.

The TRANSLATOR B is called when the deduction scheme backtracks to the last partial solution to try an alternative branch. After popping the stack, it modifies the student's SRE to allow the SELECTOR to continue. If the stack is empty, the deduction scheme halts, since all possible solutions are exhausted.

3.7.9 Error Detection in the Deduction Scheme

Below is the diagram of the deduction scheme (cf. Fig. 3-14) with the error detection portions added.

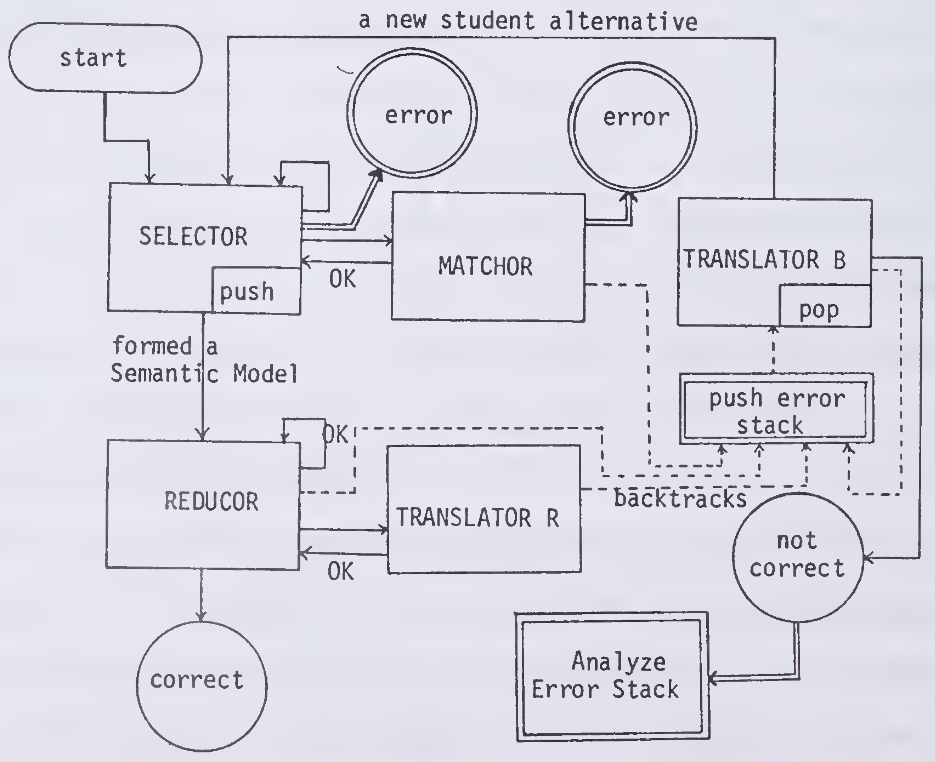


Fig. 3-38. Block Diagram of Deduction Scheme with Error Detection Portions

Situations can arise which the SELECTOR cannot resolve. The SELECTOR halts the deduction scheme and outputs an error message to the student. (Finding a loop in the student's flowchart when the instructor has an IFTHENELSE is an example of one of these situations.)

The MATCHOR can detect gross errors or inconsistencies between the student's and the instructor's flowcharts. In such cases, the MATCHOR halts the deduction scheme and outputs an error message to the student. (The unmatched conditions in a diamond may cause this, e.g., "a > 1" in the student's diamond and "b = 100" in the instructor's diamond.)

Whenever a backtrack occurs, the reason why it occurred and other pertinent facts are pushed into an Error Stack. When the TRANSLATOR B attempts to pop an empty Main Stack, the deduction scheme is halted and the Error Stack is analyzed to give the student feedback on possible errors. For example, if a backtrack is forced many times by a statement in which the MATCHOR cannot match up a constant, a possible error is that the constant has the wrong value.

3.7.10 Summary of the Deduction Scheme of PASF

The deduction scheme attempts to show that the student's flowchart is GSE to the instructor's flowchart. The deduction scheme can say that the student's flowchart is correct if it is reasonably close to the instructor's flowchart. If the deduction scheme cannot say that the student's flowchart is correct, it attempts to tell the student in intelligible language what is wrong.

The deduction scheme uses a minimum of storage. All changes by the SELECTOR, MATCHOR, REDUCOR, TRANSLATOR R, and TRANSLATOR B are done to a single copy of the student's and the instructor's SREs. An important result for the stack manager is that an original SRE can never increase during the whole deduction scheme. A quick measure of the length of the original SRE determines the size necessary for an item of the stack. A smaller flowchart can have a much larger stack depth for the same physical amount of storage. The maximum storage allowed for an SRE is 15 words. Since there are two SREs and two status words, a 32-word item may have to be pushed. Typically, an SRE is 3 words, which means that a stack item is about 8 words. The current stack size is 100 words (expandable to 1000 words). The error stack uses only one word per item; its current size is 30, which allows 30 backtracks. The total storage for data per student required for the deduction scheme is 150 words of addressable storage for work variables (TUTOR student variables) and 130 words of unaddressable storage (TUTOR STORAGE) for the two stacks. (The unaddressable storage can be transferred in blocks to the work variables.)

As noted before, the deduction scheme requires a fair amount of computing. Even though PASF is efficiently coded, a deduction may take one CPU second of computing. Depending upon the load of PLATO, one CPU second may be sixty seconds of real time. To avoid the situation in which the student becomes overly frustrated by long waits, PASF checks a real time clock at strategic places in the deduction scheme (e.g., at a backtrack), suspends operation if real time has exceeded 30 seconds, and asks the student whether he/she wishes PASF to continue. If the student types, "No," PASF returns him/her to the input routine; otherwise, PASF continues the deduction scheme for another 30 seconds of real time.

3.8 Summary of PASF

PASF allows a student to "draw" a flowchart on the PLATO screen and to have it graded. Part of the grading consists of checking to see whether it is a flowchart and, if it is, then to see whether it is a smooth flowchart. If it is a smooth flowchart, heuristic checks are performed to catch common student errors. Before the time-consuming deduction scheme is attempted, algorithm specific checks are performed to assure that the flowchart is close to the right answer. After passing these checks, the student's flowchart is compared with a correct flowchart inputted by the instructor. The deduction scheme attempts to show that the student's flowchart is correct by proving it is GSE to the instructor's flowchart. At all levels of PASF, feedback on errors is given to the student.

PASF is a viable program to grade student generated flowcharts in the PLATO computer-based education environment.

Chapter Four

THE ROLE OF THE INSTRUCTOR IN PASF

The Program to Analyze Smooth Flowcharts (PASF) is used by three classes of people--students, researchers, and instructors. The student's use of PASF is discussed in Chapter Three. The chief researcher is the author. The subject of this chapter is the role of an instructor[13] using PASF to design exercises and to check student comprehension of step-wise refinement and/or flowcharting techniques.

4.1 Design of Exercises

The design of the exercises by an instructor is an integral part of PASF. The author feels that if an educational tool such as PASF ever achieves success by being used by instructors in computer science courses, it must allow the instructors to design their own exercises. Since each instructor runs his/her courses slightly differently, he/she will want to incorporate different exercises. For example, if a "canned" set of exercises were designed by the author, then they would be used only by the author. Furthermore, allowing the instructor to change the exercises often, e.g., each term, will discourage the students from cheating. PASF

[13]The instructor's tasks are fully described in "The Instructor's Manual for PASF" [Hyde, 1975].

has been designed to handle, within a certain programming domain, any exercise dreamed up by an instructor.

4.1.1 Importance of an Instructor's Awareness of the Scope of the Programming Domain Permitted by PASF

The instructor, e.g., professor or teaching assistant in a CS100-level course, must be conscious of the limitations of PASF. Obviously, PASF will not handle a large program, e.g., a compiler. The instructor must realize that PASF is meant to be an educational tool to teach introductory nonmajor programmers in their frustrating first weeks of a programming course. The instructor must be aware that the exercises must be designed within the programming domain of PASF, i.e., the simple language SIM in up to twenty-one flowchart boxes.

4.1.2 Design of an Exercise Off-Line

The design of an exercise by an instructor involves both off-line and on-line tasks. The off-line task is writing the word problem. The instructor must be aware of several limitations in writing the word problem. First, the wording must suggest to the student a *smooth* flowchart. Secondly, the word problem should clearly specify the flowchart to be drawn by the student. The instructor should not leave pieces of the flowchart, e.g., the input and output, unspecified. Thirdly, the word problem must suggest only one algorithm, not a class of algorithms (e.g., bubble sort vs. sort) to the student. These last two restrictions arise because PASF assumes that the instructor's flowchart is Global Semantic Equivalent (GSE) (cf. Section 2.4) to the word problem. If the word problem is vague and unspecified, the instructor's flowchart cannot be GSE to it. These limitations on the instructor are not serious, but do require that he/she design the exercises carefully.

4.1.3 Design of an Exercise On-Line

After designing the word problem off-line, the instructor sits at a PLATO IV terminal to finish the design on-line. By his/her sign-on name, PASF knows that the person sitting at the terminal is an instructor. PASF gives him/her special instructions and allows options not available to a student. These instructions relate the four steps necessary to design an algorithm representing a word problem:

1. Design the acceptable English phrases;
2. Type in the name of the exercise;
3. "Draw" the correct smooth flowchart; and
4. Answer a series of questions.

Each of these four steps will be covered individually in the following sections.

4.1.4 English Phrases in Flowchart Boxes

If the instructor wants students to type English phrases in flowchart boxes, he/she must design the acceptable English phrases. For each English phrase, the instructor must insert a TUTOR "ANSWER" command or groups of ANSWER commands separated by "OR" commands at the appropriate place in PASF's code.[14] The special instructions given by PASF tell the instructor where to insert the ANSWER commands. Since the maximum number of different English phrases per exercise is four, the instructor inserts one, two, three, or four ANSWER commands, depending on the number of different English phrases he/she wants the student to input.

The ANSWER command below

Command	Tag
ANSWER	<the, are, all, of> dishes (done, finished, completed)

matches any of the following student responses:

"dishes done"

"all the dishes are finished?"

"are all the dishes completed?"

For those unfamiliar with TUTOR, the ANSWER command matches the words in its "tag" to the words in a student response. The words in the tag inside parentheses are synonyms; words inside of "<" and ">" are possible extra words. All the other words in the tag are

---

[14]"The Instructor's Manual for PASF" [Hyde, 1975] describes the use of English phrases more thoroughly.

required for a match of a student response. The same order of synonyms and important words is also required.

The insertion of ANSWER commands in PASF's code causes several logistic problems. First, the instructor must exit from PASF and edit the code. Second, the code of PASF must be recondensed (recompiled) before the instructor can reenter PASF. The first logistic problem limits the people who may be an instructor. He/she must be able to edit a TUTOR file and insert the ANSWER commands. Since a recondense of a program will "kick out" (i.e., remove from program) any students currently running in that program, the second logistic problem requires that the instructor do his/her adding of exercises at hours when students are not using PASF. Together the two problems mentioned above force the stipulation that only one instructor can input an exercise at any one time. These logistic problems are not too serious, for it is expected that exercises will be inputted, every week, not every hour. The insertions of ANSWER commands in PASF to handle English phrases is not a perfect solution, but it is acceptable.

4.1.5 The Name of the Exercise

The instructor types the name (up to ten characters) of each exercise. Since this name is used by PASF to distinguish the different exercises, the same name must appear on the paper handout given to the student.

4.1.6 <u>The Correct Smooth Flowchart</u>

The instructor inputs a correct solution in the form of a smooth flowchart. After inserting ANSWER commands (if there are English phrases) and giving a name to the exercise, the instructor "draws," in a way similar to the student (cf. Section 3.5), the correct flowchart. When the instructor finishes drawing his flowchart, he/she asks for PASF to check it. Like the student's flowchart, the instructor's flowchart must be a legal flowchart, must be a smooth flowchart, and must pass all the heuristic checks.

Since much of PASF's success in analyzing the student's flowcharts is dependent upon the instructor's flowchart, the instructor must be conscious of several facts before "drawing" his/her flowchart. First, the instructor's flowchart must correctly reflect the word problem. Second, it should be the typical solution expected from all the students. An esoteric solution from an instructor may mean the failure of PASF to handle the students' correct solutions. The instructor must realize that it is his/her responsibility to "draw" a flowchart which increases the success of PASF in deducing the students' correct solutions.

4.1.7 <u>Answers to a Series of Questions</u>

For the algorithm specific checks and the deduction scheme performed on the student's flowchart (cf. Sections 3.6.7 and 3.7), the instructor answers about a dozen questions, which are quickly and easily answered, about his/her flowchart on the screen. Examples of the questions used by the algorithm-specific checks are, "What is the minimum number of 'read's for this exercise?"; "What is the minimum number of loops for this exercise?"; and "Is a 'read' inside a loop?" Other answers inputted by the instructor are used by the deduction scheme.

An example of a question the answer to which is used by the deduction scheme is, "Is the order of 'read's important?" If the instructor demands that the "read"s ("print"s and English phrases are similar) must be in order, the deduction scheme is altered when a student's flowchart is checked. The alteration includes a special output variable added to each "read" in the Standard Regular Expression (SRE) representations. This new output variable (different from any variable a student could use) changes the performance of TRANSLATOR B and TRANSLATOR R. The two translators will not be allowed to interchange two "read"s because of the same output variable (cf. Condition 3 of Section 3.7.7).

After the instructor answers the questions, PASF checks his/her flowchart for consistency with his/her answers to the questions. If there is an inconsistency, e.g., minimum of two "print"s required but only one "print" in the instructor's inputted flowchart, PASF requires him/her to reanswer the questions. If there are no inconsistencies, PASF stores the instructor's exercise and the instructor is asked whether he/she would like to input another exercise.

4.1.8 Storage of Instructor's Flowchart and Data

As soon as the instructor's answer to the last question is accepted by PASF, the flowchart and other data are stored and ready for use by the students. Thirty words of storage is needed to store each of the instructor's exercises. Up to thirty exercises are stored and accessible to all the students as read only memory (TUTOR COMMON). Very little storage (thirty words) is required to store everything needed for an exercise.

4.1.9 Summary of Inputting an Exercise

To input an exercise, the instructor must insert the ANSWER commands, type the name, "draw" a correct smooth flowchart, and answer about a dozen questions. After a little practice, an instructor sitting at the terminal can input an exercise in less than ten minutes. Writing up the word problem will take the instructor a little longer, but since a typical exercise is less than a full typed page, the time is not excessive. The instructor does not need to know the authoring language TUTOR or the PLATO system very well. If an instructor does not use English phrases, then he/she needs no knowledge of TUTOR. PASF has been written to allow easy and quick inputting of exercises without hindering the creativity of the instructor.

4.2 Data, Collected by PASF, Allowing Instructor to Check Student Comprehension

PASF collects data which allows an instructor to check the comprehension and progress of his/her students. The data includes flowcharts "drawn" by the students, the time the student takes to complete a section, and unanticipated student responses.

PASF collects data which is sufficient to recreate a student's flowchart. A copy of every flowchart that a student checks is placed into a special file (PLATO data file). PASF transfers sixty-six words per copy of flowchart. (This allows up to 65 copies of flowcharts in a PLATO single part data file.) At his/her leisure, e.g., a week later, the instructor can retrieve the data and recreate the student's screen at the state when the student asked for his/her flowchart to be checked by PASF. For the instructor's convenience, the name of the student and the number of the flowchart in the data file are displayed along with the student's copy of the flowchart. In Fig. 3-6 of Chapter Three is a copy of "maryjane's" attempt at algorithm "intdivide," displayed several days after she drew the flowchart. The "3" in the upper right hand corner in Fig. 3-6 signifies that this is the third flowchart in the data file. It should be noted that the array of boxes and arrows in Fig. 3-6 is identical to the flowchart maryjane saw, and not a topological transformation of her flowchart. Not only can the instructor see a copy of the student's flowchart, but he/

she can also "check" the flowchart and observe the error message, if any, PASF gave to the student, and can modify the copy. These capabilities of seeing, checking, and modifying a copy of a student's flowchart allow the instructor to judge the abilities of his/her students.

In the data file PASF collects unanticipated student responses which were not matched by PASF. Of interest to an instructor are the unmatched student attempts at the English phrases. After the first few students have done an exercise, the instructor reads the unmatched student attempts in the data file concerning his/her English phrases and revises his/her ANSWER commands (cf. Section 4.1.4). This refinement resulting from unaccepted student responses is important in any computer-based educational program.

It is clear that the data collected by PASF allows an instructor to see, check, and modify a student's flowchart. These features, along with the record of the times for the student to finish each section, allow an instructor to monitor the performance and progress of his/her students. Data is collected to allow an instructor to increase the range of accepted student versions of English phrases. The data collection features of PASF are an important consideration in the acceptance of PASF as a viable computer-based educational program.

Chapter Five

DETAILS OF FIVE ALGORITHMS IN PASF

5.1 Introduction

This chapter is a collection of PASF's algorithms, the details of which are not covered elsewhere in the thesis. To allow the discussions in Chapter Three to be readable and not burdened by details, these algorithms have been assembled in this chapter. Chapter Five discusses five algorithms:

1. The Connector Algorithm which draws the lines between boxes,
2. The algorithm to check if a legal flowchart,
3. The algorithm to check if a flowchart is smooth,
4. The algorithm which generates the Regular Expression (RE) Representation of a flowchart, and
5. The algorithm to check if two strings in the Standard Regular Expression (SRE) representation can be interchanged.

5.2 Connector Algorithm

When a student wants an arrow drawn between two boxes on the screen, he/she touches the "from" and "to" boxes and PASF automatically computes the route and draws the line. Since there may be twenty-one boxes on the screen and the line can go from any one box to any other box, the routing algorithm is much more involved to write than the author originally thought. Certain cases fail for most hastily designed approaches.

The difficulty of the problem can be comprehended if one considers the boxes as a twenty-one node graph. Fortunately, this graph is not a complete graph, since only a maximum of two arrows may leave a box (diamond). The worst case consists of nineteen diamonds and two ovals in which forty lines must be routed (19 x 2 + 2). Considering the physical limitation of the screen (512 dots by 512 dots and 8.5" by 8.5") and the twenty-one boxes (a box is 32 dots by 128 dots) already taking a large portion of the screen, one finds it difficult when drawing lines not to draw one line on top of another. The example below shows one line on top of another.

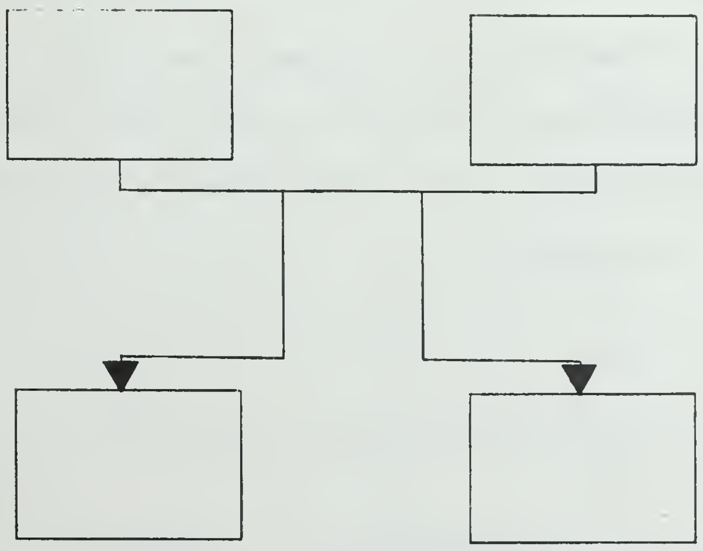


Fig. 5-1. The Problem of Two Lines on Top of Each Other

A further problem of the designer is to make the connectors appear natural. Not only must all the lines be distinguishable (e.g., not on top of each other), but they must also be close to the way a human programmer would draw them. In Fig. 5-2, the example on the left more nearly illustrates this way.

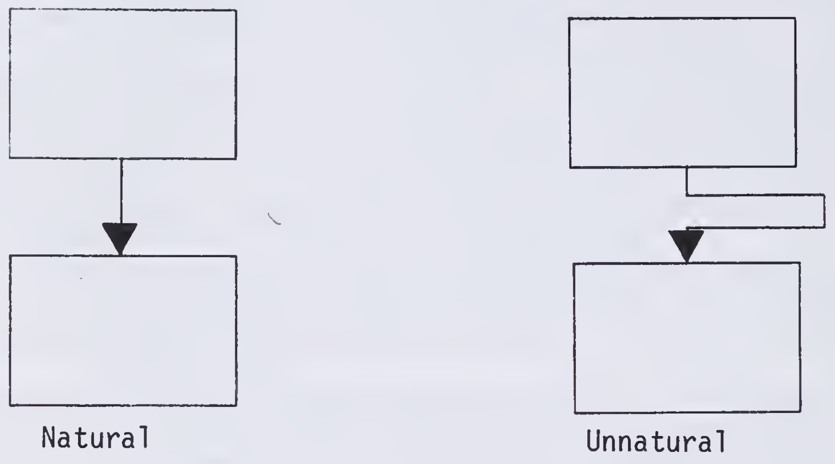


Fig. 5-2. Two Correct Ways to Draw a Line between Two Boxes

Since a student may draw the boxes in any order, the Connector Algorithm must allow for boxes to be added later. Therefore, the Connector Algorithm never draws a line through a box or through a place where a box may appear.

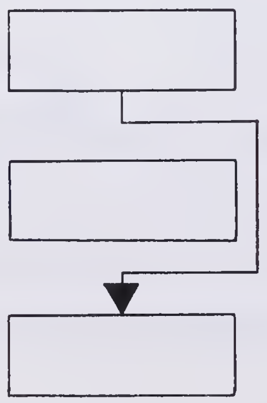


Fig. 5-3. Lines Must Never Travel Through a Box

The Connector Algorithm has been rewritten several times in order to improve the distinguishability and naturalness. The current version of the algorithm follows.

The "from" box is designated by coordinates (x, y). The "to" box is designated by coordinates (w, z). Since there are 21 specific positions of three columns of seven, x and w range from 1 to 3, and y and z range from 1 to 7. Below is shown the coordinate system for the Connector Algorithm.

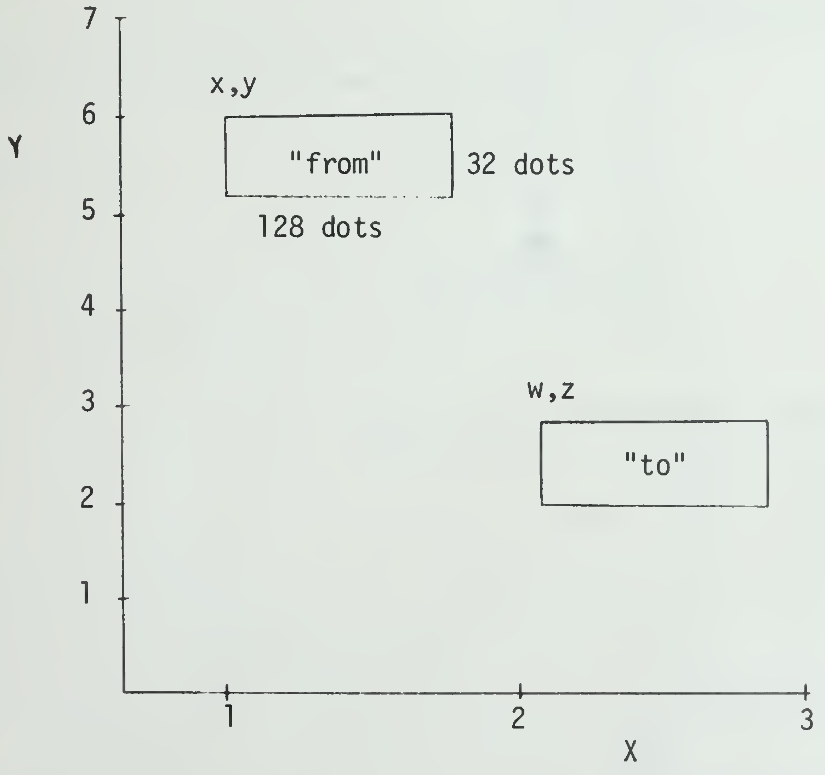


Fig. 5-4. X, Y Coordinate System for the Connector Algorithm

The positions of the two boxes fall into five cases. Case one is tested first. If case one fails, case two is tested, etc. If a case applies, the line is drawn as shown. In Figs. 5-5 through 5-9, some lines are labeled with their lengths in dots and some lines are functions of "y" or "w" dots.

Case 1: x = z and x + 1 = w

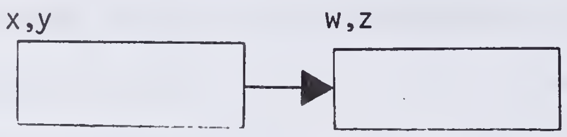


Fig. 5-5. Case 1 of Connector Algorithm

Case 2: x = w and y = z + 1

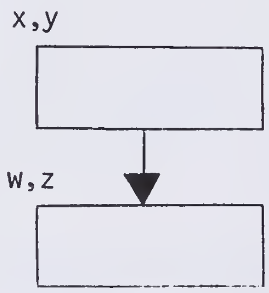


Fig. 5-6. Case 2 of Connector Algorithm

Case 3: abs (x - w) = 1 and y = z + 1

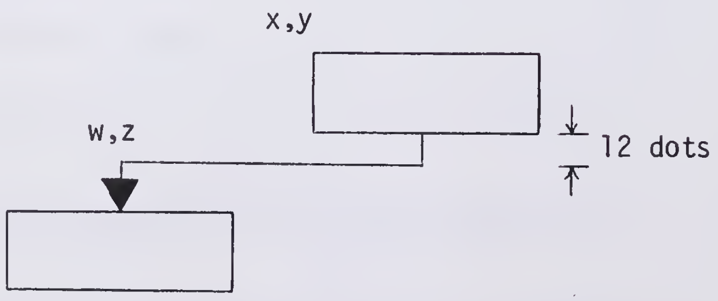


Fig. 5-7. Case 3 of Connector Algorithm

Case 4: $y \geq z$

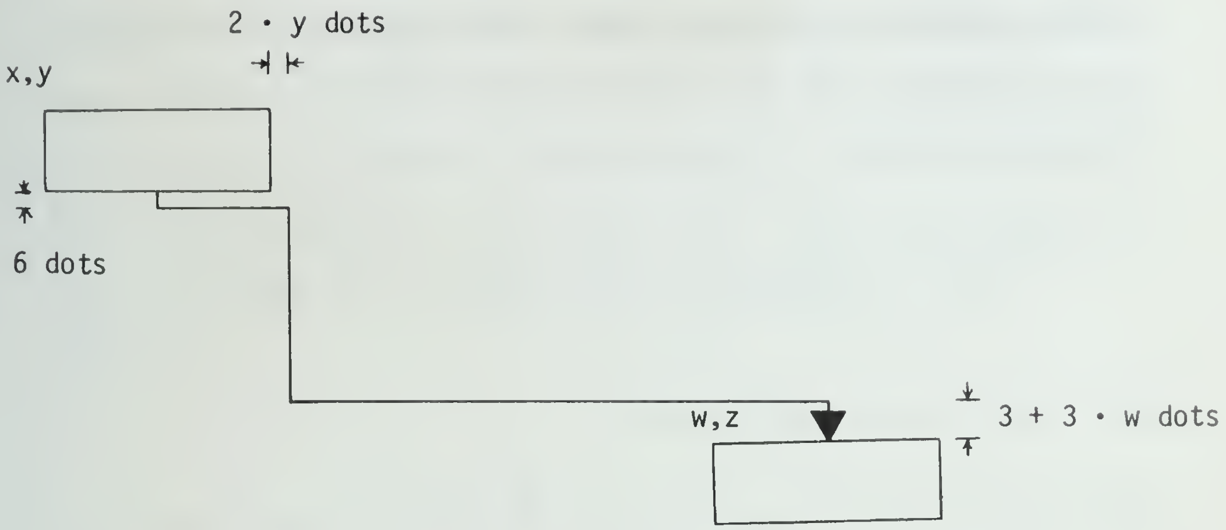


Fig. 5-8. Case 4 of Connector Algorithm

Case 5: $y < z$

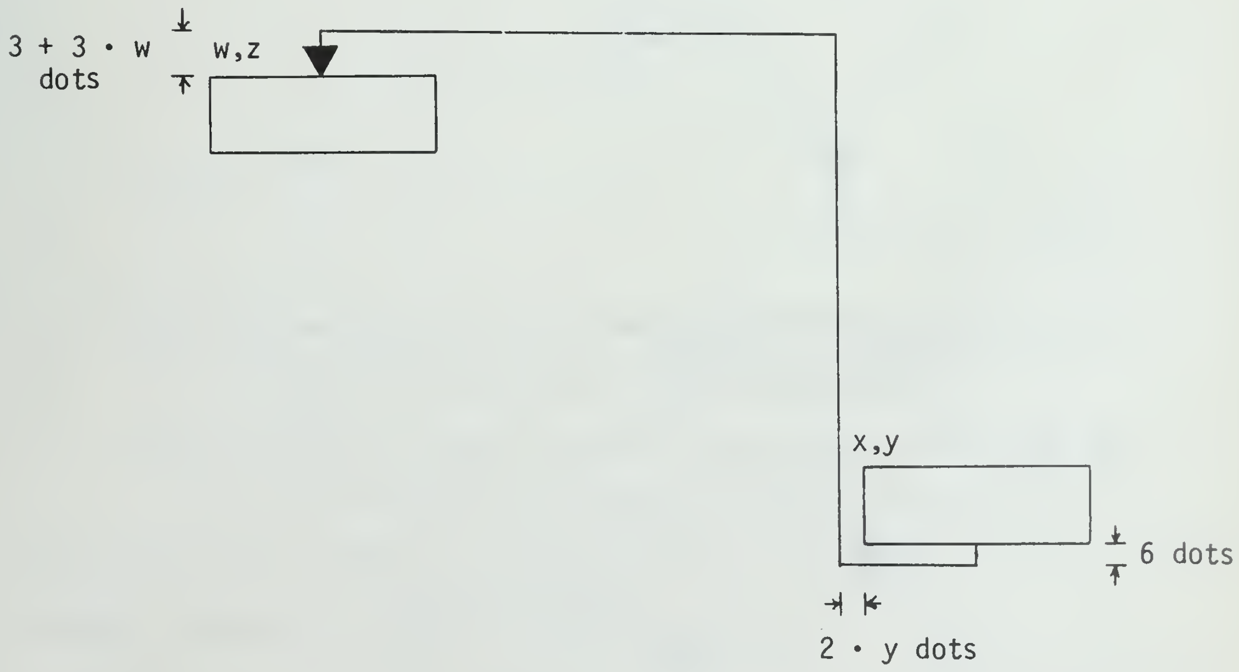


Fig. 5-9. Case 5 of Connector Algorithm

It should be noted that cases 1, 2, and 3 follow the flowchart convention that boxes should tend to flow from the top to the bottom and from the left to the right. The case in Fig. 5-10 is not allowed,

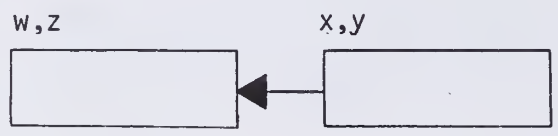


Fig. 5-10. Not Allowed in PASF

since the situation in Fig. 5-11 with two arrows lying on top of each other arises.

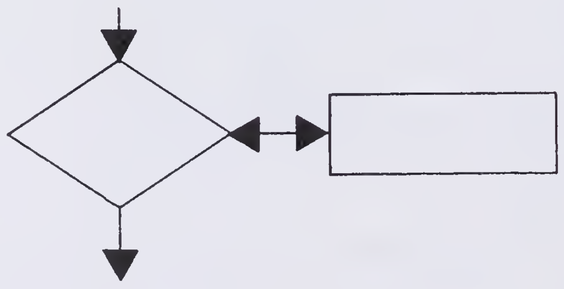


Fig. 5-11. An Undesirable Situation of Two Lines on Top of Each Other in a DOWHILE

The undesirable example above would be drawn by PASF as follows:

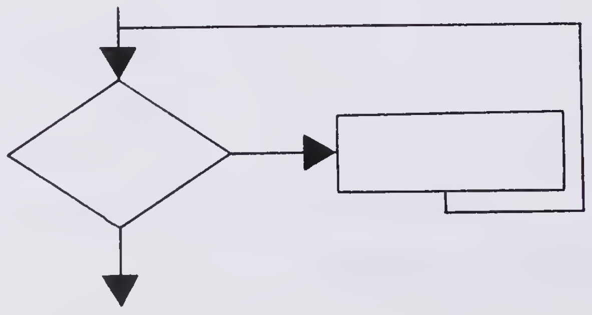


Fig. 5-12. The Example of Fig. 5-11 Drawn by PASF

The five cases illustrated above are not sufficient to enable PASF to draw the lines. The diamonds must be handled in a special way. Since two arrows are drawn from a diamond, PASF draws the first by the above five cases and draws the second line from the right side (Case 6) if the first line is drawn from the bottom. The "T" and the "F" are plotted near the appropriate line.

<u>Case 6</u>:

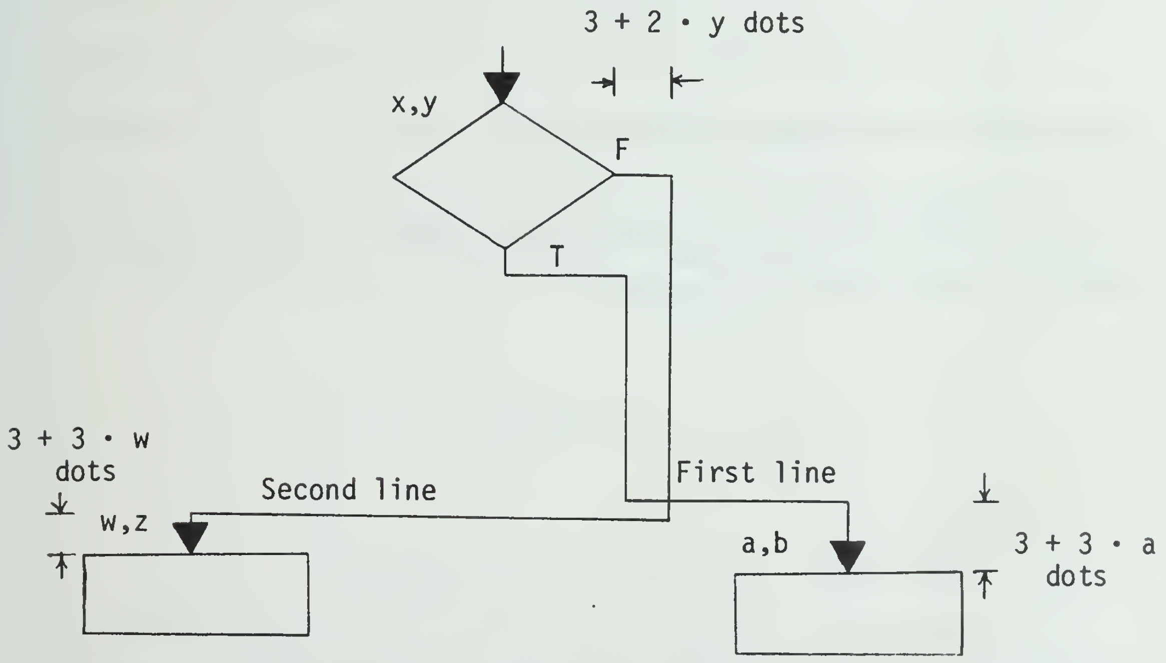


Fig. 5-13. The Second Arrow of the Diamond is Drawn to the Right as Shown if First Arrow is Drawn from Bottom

The lines drawn by the Connector Algorithm are shifted a few dots to allow them to be distinguishable. There are 31 dots between adjacent horizontal or vertical boxes.

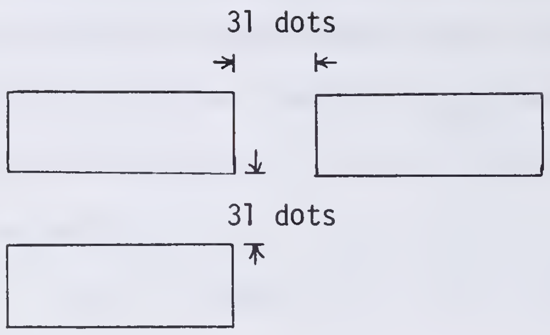


Fig. 5-14. Dots between Adjacent Boxes

The vertical case in which horizontal lines must be drawn is divided into two regions:

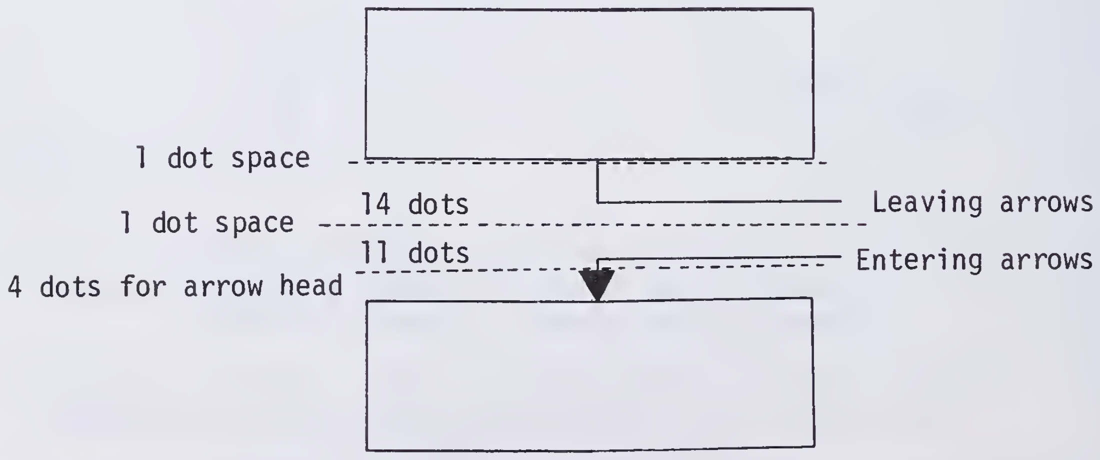


Fig. 5-15. Regions for Horizontal Lines between Two Adjacent Boxes

There are only two ways for "leaving arrows":

1. Case 3, which uses 12 dots from the bottom of the box, and
2. Cases 4 and 5, which use 6 dots from the bottom of the bottom of the box.

The three different "entering arrows" of the horizontal line are displaced by a function of the X coordinate of the "to" box (in Cases 4, 5, and 6). If the X coordinate is 1, 2, or 3, the "entering arrow" horizontal line is 6, 9, or 12 dots above the "to" box respectively. To displace the horizontal lines of the entering arrow, the lines are a function (3 + 3 • X) of the X coordinate of the entering box. The diagram below demonstrates this displacement.

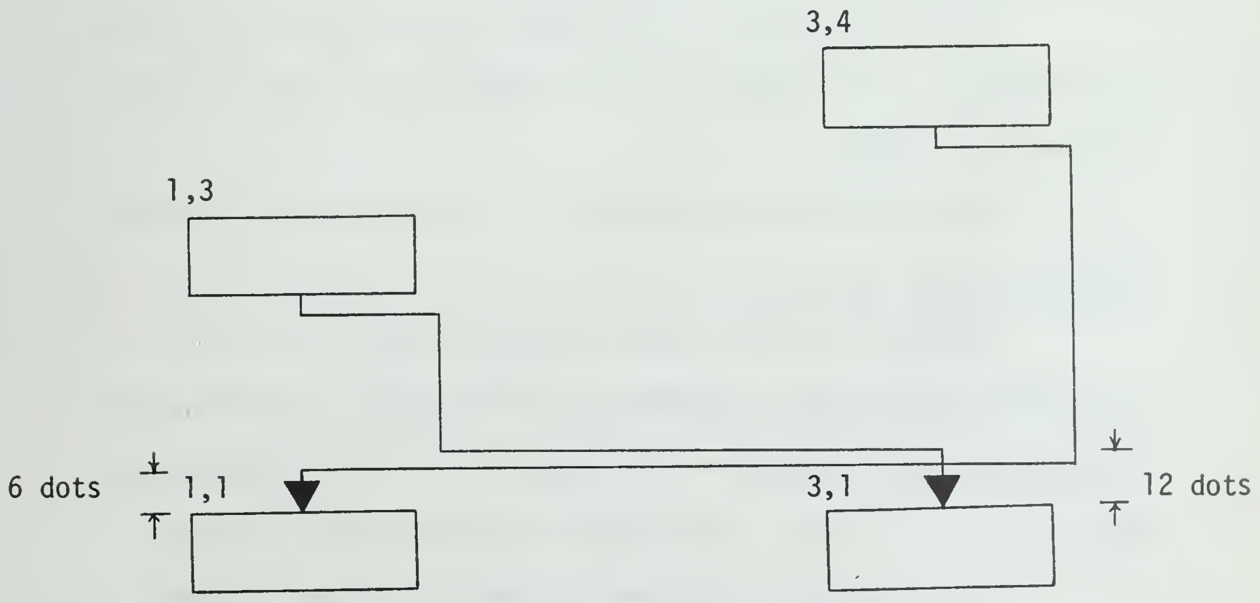


Fig. 5-16. Example to Demonstrate Displacement of "Entering Arrows"

The regions for vertical lines are divided into arrows used by Case 5 and arrows used by Case 4, as shown below.

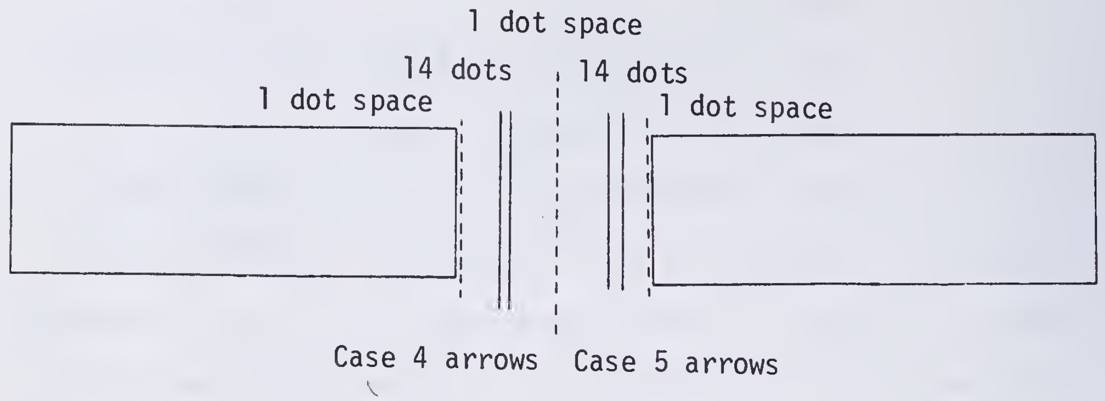


Fig. 5-17. The Regions of Vertical Lines

The function 2 • y where y = 1, 2, 3, 4, 5, 6, or 7 displaces the seven possible case 4 arrows for any column. This allows one space between each of the seven lines.

The six possible vertical Case 5 arrows (for any columns) are handled by the function 2 • y in a way similar to the way Case 4 arrows are handled.

The Case 6 vertical arrows are inserted between the Case 4 arrows by the function 3 + 2 • y.

The above routing scheme guarantees that no line lies on top of another line. The worst case is a vertical line lying adjacent to another vertical line (e.g., Case 4 arrow from rectangle at (7, 1) and Case 6 arrow from diamond at (6, 1)).

The Connector Algorithm is a feat of engineering requiring careful routing of the lines to keep them distinguishable

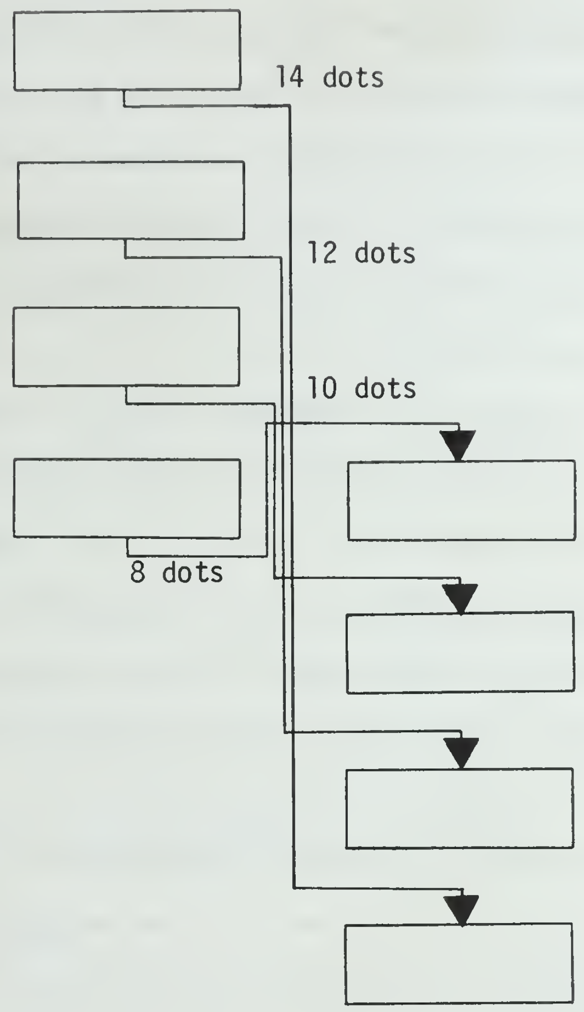


Fig. 5-18. Example to Show Displacement of Case 4 Vertical Arrows

from each other. The Connector Algorithm sacrifices slightly on naturalness, but is a viable method of drawing the arrows between the flowchart boxes.

5.3 Algorithm to Check the Legality of a Flowchart

This section discusses the algorithm which will check to see whether a student's array of arrows and boxes is a legal flowchart. A legal flowchart is defined as one (a) which has one "start," (b) which has at least one "stop," and (c) in which every box can be reached by traversing the array of boxes and arrows beginning at "start." A further qualification of a legal flowchart is that (d) text must be inside every box (cf. Section 3.6.1).

The first part of the algorithm checks to see whether the student's flowchart has one and only one "start" and at least one "stop." A "start" is recognized as a node of "oval" type which has a link leaving, no links entering, and the text "start" inside. A "stop" is recognized as a node of "oval" type which has no links leaving and the text "stop" inside. The next portion of the algorithm checks that all nodes of "rectangle" type have a right link (RL) and that all nodes of "diamond" type have both a right link (RL) and a left link (LL). With the above two portions completed, PASF begins the portion of the algorithm which traverses the flowchart.

The main portion of the algorithm traverses the list structure representation, marking every node visited. It takes the "T" side of the diamonds first and pushes into the stack any diamond found. When the algorithm hits a marked node or a "stop," the algorithm pops the stack and then takes the "F" side of the popped diamond. A one-bit flag in each node is used as the mark.

If a box can be reached from "start," the traverse will mark the node. This portion of the algorithm is well-known and easily derived from Knuth [Knuth, 1973]. Figure 5-19 presents the flowchart of the traversal portion of the algorithm.

After the traversal portion is completed, all used nodes must be marked, otherwise the array of arrows and boxes is not a flowchart. A quick check reveals any unmarked used nodes and an appropriate error message is given to the student.

The last portion of the algorithm tests for text inside every used node.

If the student's array of arrows and boxes fails in any portion of the legal flowchart checker, the student is given an appropriate message and required to fix his/her flowchart.

5.4 Algorithm to Check Smoothness of a Flowchart

5.4.1 Introduction

The Smooth Checker is a recursive algorithm based on the web grammar W_2 (cf. Section 2.2.4) which checks whether a flowchart is a smooth flowchart or not. For the reader's reference, the six rewrite rules for the web grammar W_2 are repeated below. P, b, d, and j are abbreviations for Process, box (rectangle), diamond, and join (meeting of two arrows), respectively. Rewrite rules R1 and R2 allow a series of rectangles in a smooth flowchart; R3 and R4 allow IFTHENELSEs; R5 and R6 allow the DOWHILE. The main thought

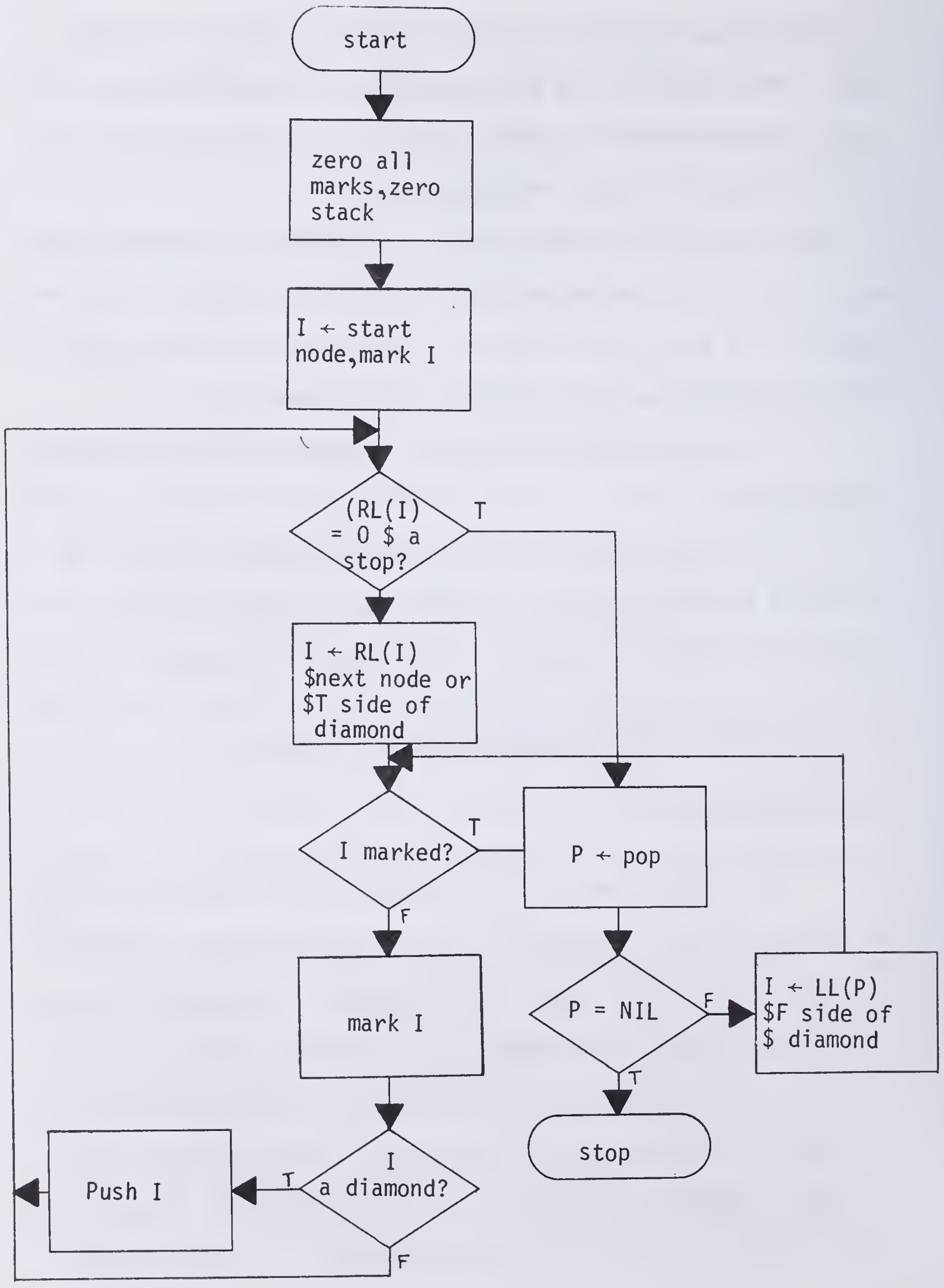


Fig. 5-19. Traversal Portion of Flowchart Checker ($ Signifies a Comment)

web, $I = \{s \rightarrow P \rightarrow e\}$; $V_N = \{P\}$; $V_T = \{s, e, b, d, j\}$

R1: $P: = P \rightarrow b$

R2: $P: = b$

R3:
$$\begin{tikzcd}
 & & P \arrow[dr] & \\
P: = P \arrow[r] & d \arrow[ur] \arrow[dr] & & j \\
 & & P \arrow[ur] &
\end{tikzcd}$$

R4:
$$\begin{tikzcd}
 & P \arrow[dr] & \\
P: = d \arrow[ur] \arrow[dr] & & j \\
 & P \arrow[ur] &
\end{tikzcd}$$

R5:
$$\begin{tikzcd}
P: = P \arrow[r] & j \arrow[rr] & & d \arrow[dl] \\
 & & P \arrow[ul] &
\end{tikzcd}$$

R6:
$$\begin{tikzcd}
P: = j \arrow[rr] & & d \arrow[dl] \\
 & P \arrow[ul] &
\end{tikzcd}$$

Fig. 5-20. Six Rewrite Rules of Web Grammar W_2

underlying the algorithm is to attempt to find the processes (P's) by looking for structures in the list structure representation which correspond to the six rewrite rules. Finding a P for a series of rectangles (R1 and R2) is fairly easy. Finding a P for an IFTHENELSE requires starting at a diamond (d); finding a P on the "T" side of IFTHENELSE upon finding a j; finding another P on the "F" side of IFTHENELSE upon finding the same j. Since a DOWHILE can loop on either "T" or "F," there are two possible cases for a P for a DOWHILE. Finding a P for a DOWHILE requires starting at a diamond (d), finding a P on the looping side ("T" or "F"), and finding a j which is directly in "front" of the diamond (d). The subsections of Section 5.4 will discuss the algorithm which recursively searches for concatenations of processes, IFTHENELSEs, and DOWHILEs in the student-generated flowchart.

5.4.2 Problems Due to Differences in the Web Grammar and List Structure Representations of Flowcharts

Since there are differences between the web grammar and the list structure representations of a flowchart, an algorithm which operates on the list structure should not simulate exactly the six rewrite rules of the web grammar. The differences between the web grammar and the list structure representations are two:

1. A null rectangle can exist in the list structure representation; and
2. Adjacent j's in the web grammar exist as only one j in the list structure.

As shown below, the first difference of a null rectangle can exist in a smooth flowchart as well.

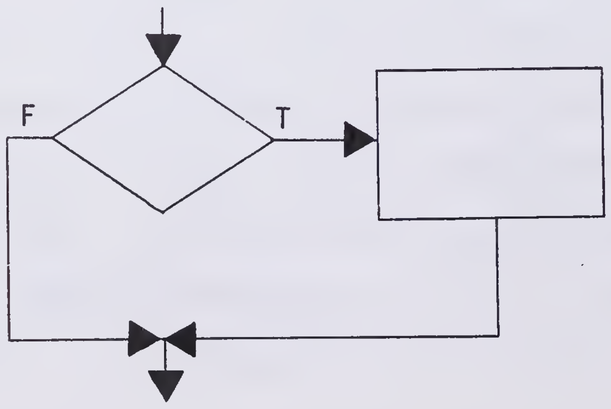


Fig. 5-21. A Null Rectangle on the F-Side of an IFTHENELSE

The occurrence of null rectangles causes no difficulty with the Smooth Checker. Figure 5-23 shows a flowchart with its web grammar representation and list structure representation to demonstrate the possible difference in the number of j's. In the list structure, a j is any node with more than one link pointing to it. A one-bit field is set in the node to designate that the node is a j. (In the list structure below and on the following pages, a node in the list structure has the following fields:

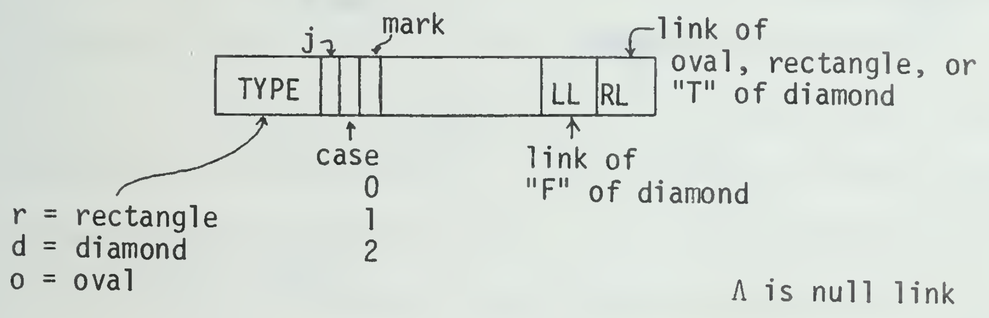
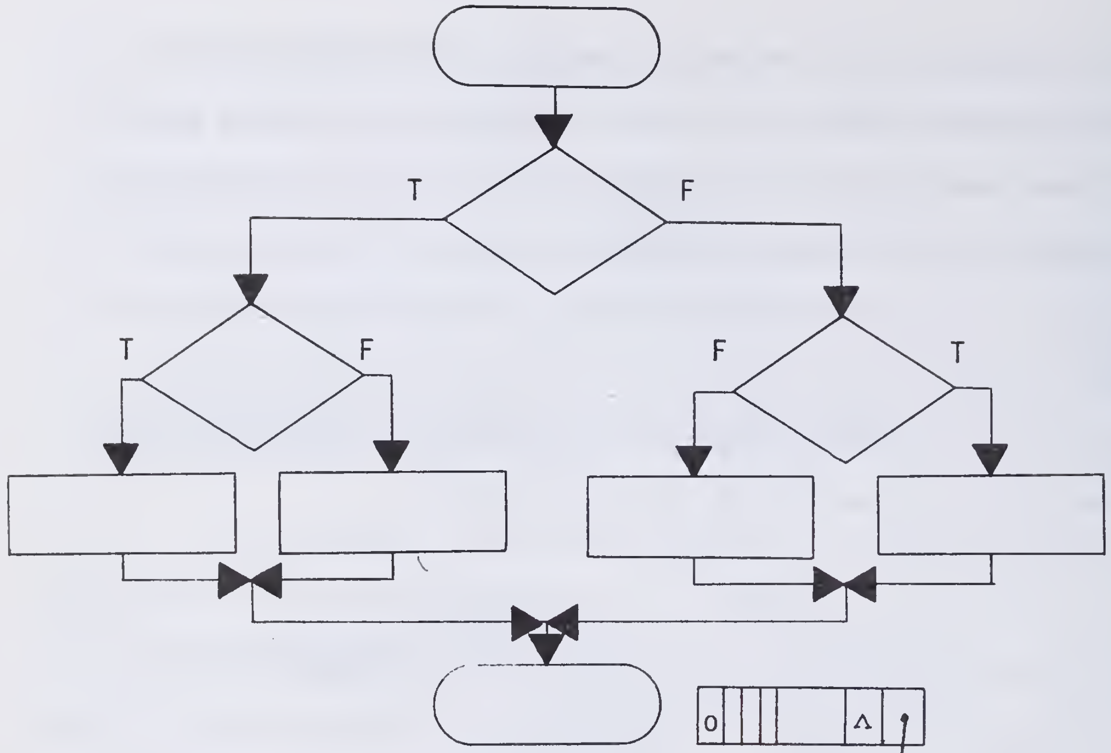


Fig. 5-22. The Fields in a Node in the List Structure

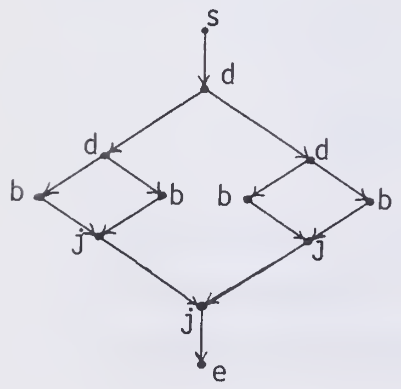
where the possible values of the fields are shown.) Since all four links point to the "oval" node, the three j's in the above web grammar are no longer distinct in the list structure. The j's do cause complications in the Smooth Checker. The list structure always has a j where it is necessary, but the j may be required to be used many times.

5.4.3 Marking the J's in the List Structure

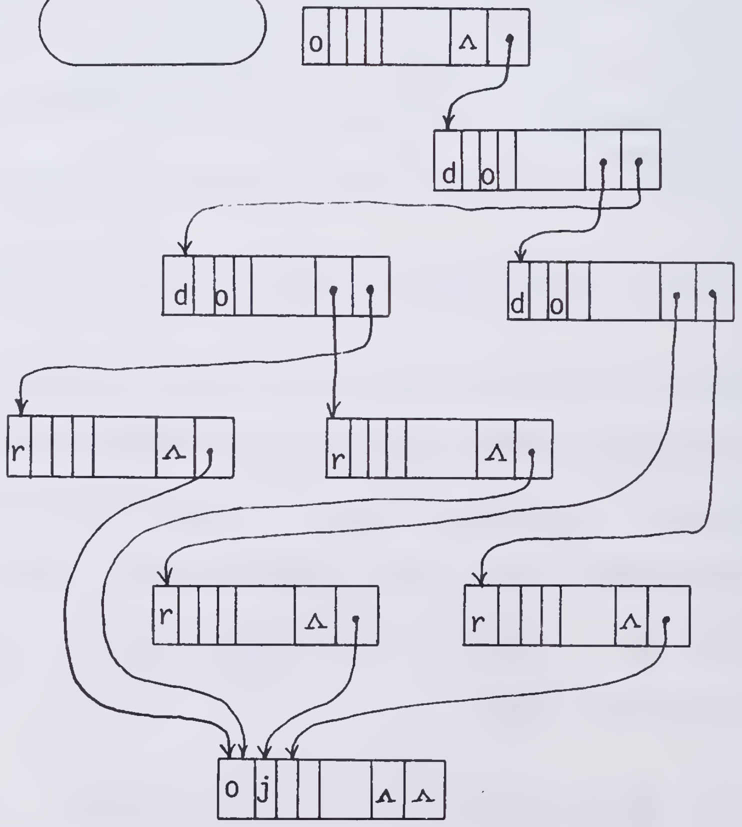
A j in the web grammar is associated with a one-bit field



Flowchart A



Web Grammer of
Flowchart A



List Structure of Flowchart A

Fig. 5-23. Smooth Flowchart A with Embedded IFTHENELSE and Its Web Grammar and List Structure Representations

in some node of the list structure. If any node has more than one link pointing to it, the node is a j and the j field is set. The setting of the j field is accomplished with very little extra computation during the execution of the Flowchart Checker (cf. Section 5.3).

The Flowchart Checker which checks whether the student's array of arrows and boxes is a flowchart sets the j field. A portion of the Flowchart Checker traverses the list structure marking each node it visits. If it visits a node already marked, the Flowchart Checker pops a stack and continues on a different link (LL) of a diamond node. Every node that is visited and already marked must be a j, since the Flowchart Checker reached that node by more than one different path. The Flowchart Checker sets the j flag whenever it visits a node that is already marked.

5.4.4 Brief Description of the Smooth Checker

Before the Smooth Checker begins, it is known that the list structure is a legal flowchart and the j fields have been set. The Smooth Checker is divided into two routines, SR1 and SR. SR1 is the main routine which calls SR every time a diamond node at the zero level (an unembedded IFTHENELSE or DOWHILE) is found. Routine SR calls itself recursively after finding an embedded diamond node. SR first tries to find whether a diamond is part of an IFTHENELSE. If the diamond cannot be an IFTHENELSE, SR tries to show that the diamond is part of a DOWHILE which loops on "T." If the diamond is not a DOWHILE which loops on "T," SR tries to show that the diamond is a DOWHILE which loops on "F." If this previous case fails, the flowchart cannot be smooth and an appropriate message is given to the student.

The algorithm for the Smooth Checker will work on an arbitrary flowchart with arbitrary embedding of DOWHILEs and IFTHENELSEs. PASF's implementation is limited to a depth of three embedded DOWHILEs and IFTHENELSEs, e.g., a DOWHILE inside an IFTHENELSE inside a DOWHILE inside an IFTHENELSE. PASF's limitation of a depth of three embedded constructs is because of a stack memory limit. This limitation is not restrictive, for PASF only allows twenty-one boxes.

When SR is trying to show that a diamond is an IFTHENELSE, SR traverses the "T" side and finds a j. SR places a pointer at the j and traverses the "F" side of the diamond until SR finds a j and sets a pointer. If the first j is the same node as the second j, SR has found an IFTHENELSE and returns to where it was called. Either j, because of embedded IFTHENELSEs and DOWHILEs, may not be the correct j for the diamond. SR systematically moves across the embedded diamonds the two pointers pointing at the two j's until the two pointers point at the same node. If all the possible ways for the two j's are tried and have all failed, SR tries to show that the diamond is part of a DOWHILE.

For a DOWHILE, SR traverses the side which loops (first the "T" side, then the "F" side) and finds a j. If this j is the same node as the diamond, then SR has found a DOWHILE and returns to where it was called. If the j is not the same node as the diamond, the j is assumed to be a part of an embedded DOWHILE or IFTHENELSE. SR continues on to the next j and checks to see whether the node is the diamond. If the j fails to be the diamond, SR tries the next case (e.g., DOWHILE which loops on "F"). If the DOWHILE which loops on "F" fails, SR declares the flowchart unsmooth and gives the student the appropriate message.

Every time SR encounters an embedded diamond, the appropriate pointers and lists are pushed into the stack and SR is called recursively. Upon finding that the embedded diamond is smooth, SR returns the call, pops the stack, and continues.

The next section will cover the details of the main routine SR1. The details of the recursive routine SR follows in Section 5.4.6. A discussion of the problems involved in always having the Smooth Checker terminate appears in the last section of 5.4.

5.4.5 Details of SR1 Routine

Figures 5-24 and 5-25 show the flowchart for the routine SR1. The discussion in this section as well as in the next section will require the reader to glance often at these flowcharts.

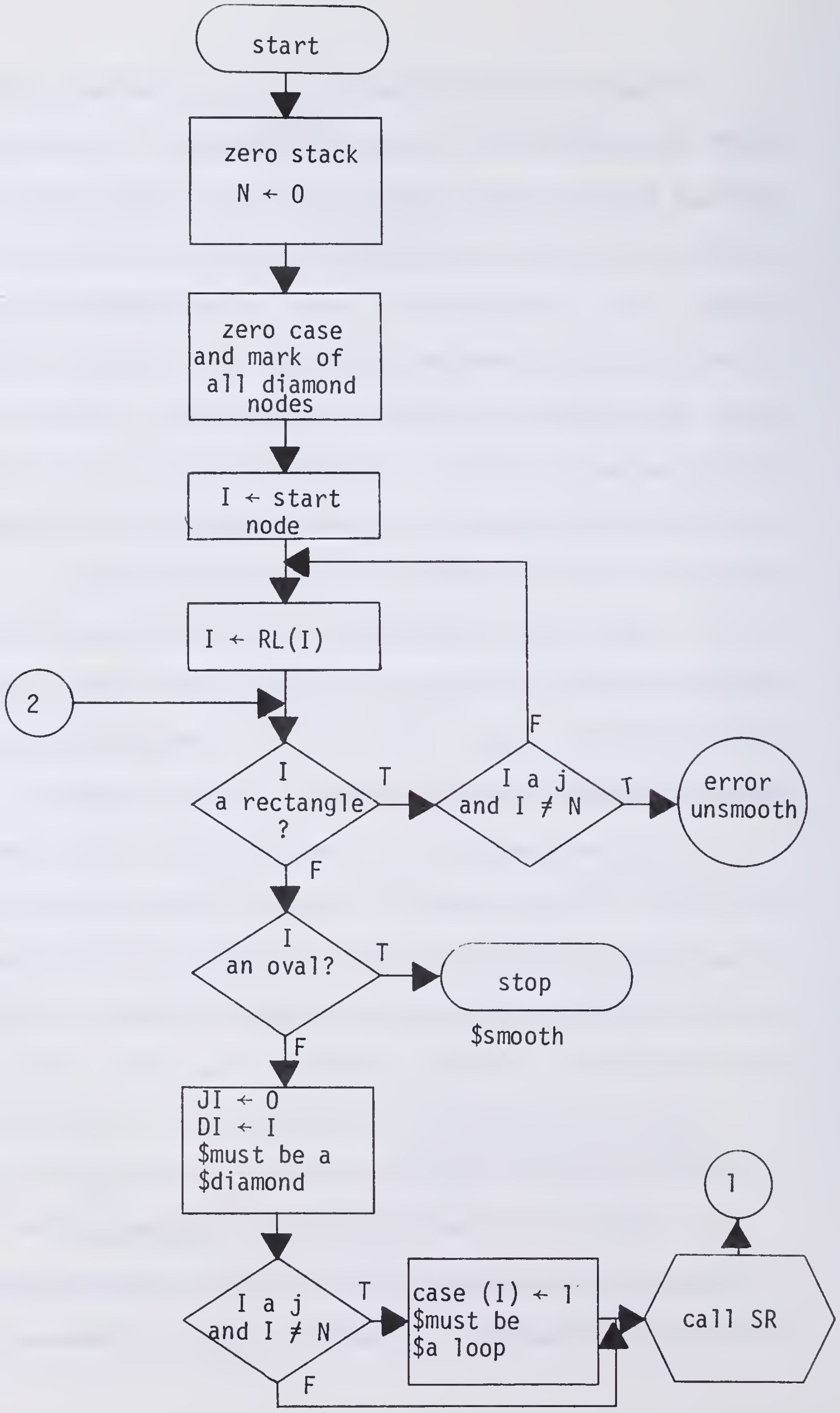


Fig. 5-24. Part I of Flowchart for Routine SR1

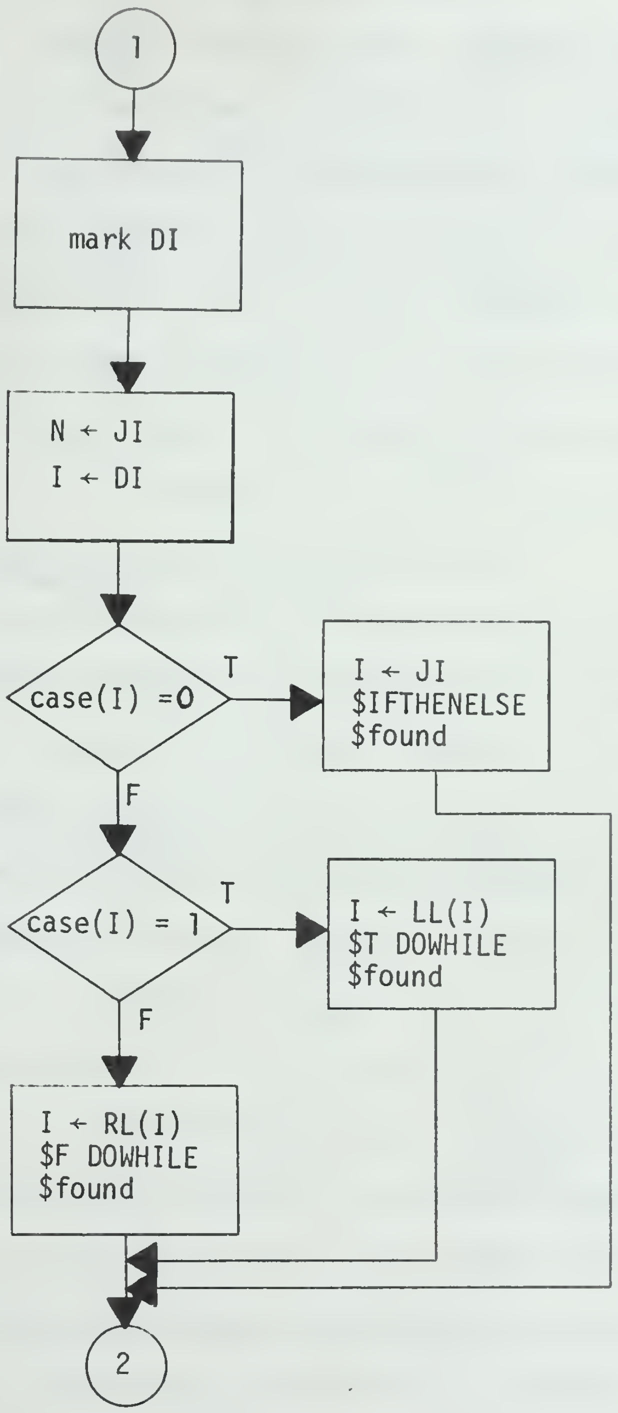


Fig. 5-25. Part II of Flowchart of Routine SR1

In the routine SR1 (cf. Fig. 5-24), the stack and N are zeroed and the "case" and "mark" fields in every diamond node are zeroed. To be smooth, a diamond must be part of an IFTHENELSE, a DOWHILE which loops on "T," or a DOWHILE which loops on "F." Routine SR, which is called by SR1, assumes a diamond is part of an IFTHENELSE (case = 0) first. If SR shows that this diamond cannot be part of an IFTHENELSE, then SR tries a DOWHILE which loops on "T" (case = 1). If case = 1 fails, then a DOWHILE which loops on "F" (case = 2) is tried.

Routine SR1 (cf. Fig. 5-24) begins by setting I (pointer of the current node with a value from 1 to 21) to the "start" node and visits the next node ($I \leftarrow RL(I)$). If the next node is a rectangle, a special check (to be discussed later) is performed to catch a class of errors, and the next node is visited. If the next node is not a rectangle and is an oval, the SR1 is finished and the flowchart is smooth. Assuming the next node is not a rectangle or an oval, it must be a diamond. JI (pointer to a node which is a j) is set to zero and DI (a pointer which points at the current diamond node) is set to I. If I is a j and $I \neq N$, then this diamond cannot be an IFTHENELSE (the reason for this will be discussed later) and case (I) is set to one. SR1 calls SR. If SR returns to SR1, SR has succeeded in showing that the diamond DI is a DOWHILE or an IFTHENELSE and has set JI and the "case" of DI. Below are the three flowchart segments on which SR will succeed (assuming "a," "b," "c," and "e" are smooth).

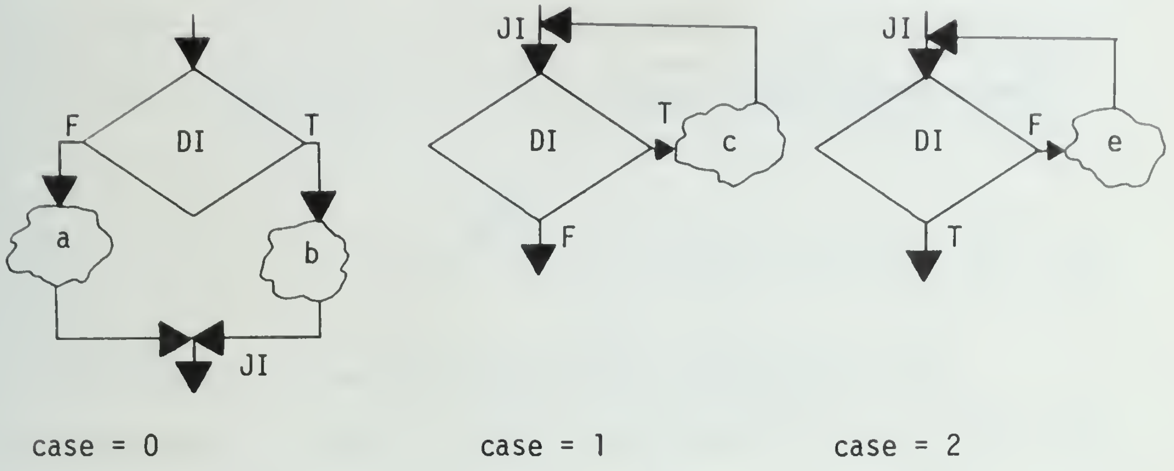


Fig. 5-26. Three Cases in which SR Will Succeed

Below are the list structure representations of these three cases.

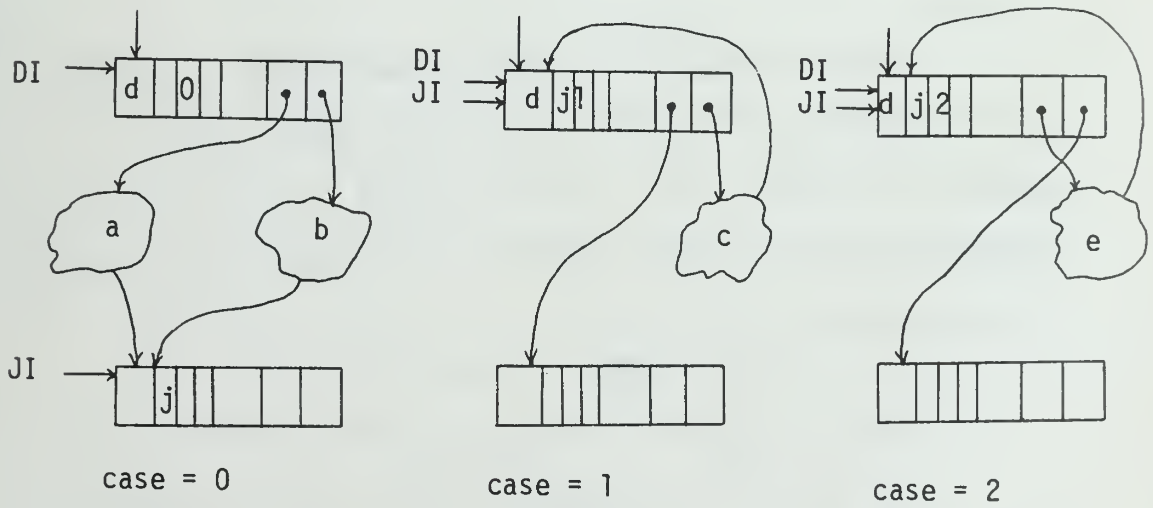


Fig. 5-27. List Structure Representations for the Cases of Fig. 5-26

The fields in a node are again shown below.

TYPE	JFLAG	CASE	MARK		LL	RL

Fig. 5-28. Fields in a Node

After the return by SR, routine SR1 marks DI "finished" and continues by visiting the next node. If case = 0, then the next node is the one pointed to by JI. If case = 1, then the next node is the one pointed to by the left link (LL) of the diamond. If case = 2, then the next node is the one pointed to by the right link (RL) of the diamond. Routine SR1 continues until it finds an oval node.

The two discussions deferred above involve the pointer N. N is equal to the last JI. In the first check, node I is a rectangle. If I is a j and I = N, then the j is the result of an IFTHENELSE SR found directly "before" the rectangle and terminated on the rectangle (cf. Fig. 5-29).

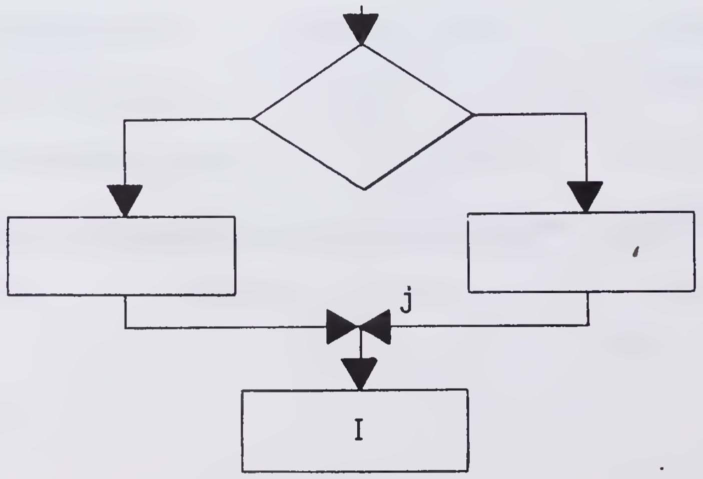


Fig. 5-29. A Rectangle is a J Because of the IFTHENELSE "Before"

The above situation is smooth and no error occurs. If I is a j and I ≠ N, then the j must be the result of an arrow of a bad loop. The arrow cannot be from an IFTHENELSE, for then I would equal N.

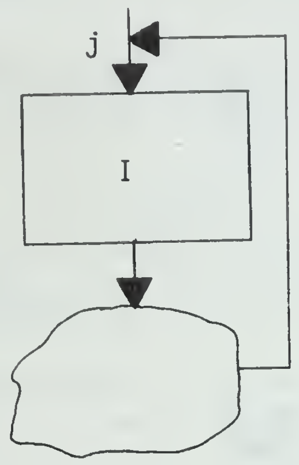


Fig. 5-30. A Bad Loop, Since the Rectangle is a J and I ≠ N

The flowchart segment above cannot be a DOWHILE, since the j corresponding to a DOWHILE is always a diamond node. Therefore, it must be a bad loop. A message is given to the student saying the flowchart is unsmooth because the arrow pointing to the rectangle causes a bad loop.

The second check involves a diamond which is a j. As with the first check, the j might be a result of an IFTHENELSE SR found directly "before." If an IFTHENELSE is "before," then I = N. If I ≠ N, then an IFTHENELSE is not "before," and the diamond must be part of a DOWHILE loop or part of something unsmooth. To

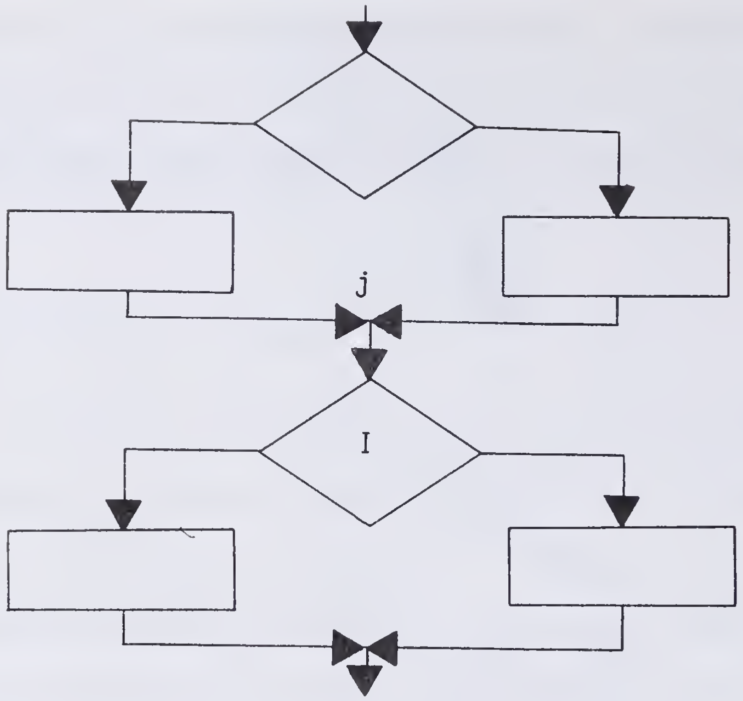


Fig. 5-31. A Diamond which is a J Might be an IFTHENELSE if I = N

communicate to SR the message not to try the case of an IFTHENELSE, SR1 sets the "case" of DI equal to one.

Routine SR1 handles all the level-0 rectangles and calls SR for every level-0 diamond. If SR1 finds an oval, SR1 halts and tells the student his/her flowchart is smooth.

5.4.6 Details of the Routine SR

Routine SR, which is called by SR1 and itself, tries to show that a diamond is part of an IFTHENELSE, a DOWHILE which loops on "T," or a DOWHILE which loops on "F." These three cases are tried in the above order. If a case fails, the next case is tried.

If none of the three cases succeed, SR halts and tells the student that his/her flowchart is unsmooth and points out the offending diamond. In certain situations, SR will determine that the flowchart is unsmooth and will not try the next case.

The routine SR is shown as a flowchart spread over five figures (Fig. 5-32 through Fig. 5-36). Since all three cases are shown together, the routine is complex.

Part of Fig. 5-32 shows an outer loop which increments case (DI) by one and tries the complete routine again. If case (DI) = 3, then all three cases have been attempted and the student's flowchart is unsmooth. If a new case is tried, JI is set to zero, the diamonds in TLIST and FLIST (marked while trying the old case) are unmarked, TLIST and FLIST are zeroed, LASTTJ and LASTFJ are set to zero, and a new I is selected, depending on case (DI). DI points to the diamond node that SR is trying to show as smooth or not smooth. I points to the current node. JI points to a j. TLIST and FLIST are two lists of all the diamonds that are marked while attempting to show that DI is smooth. For an IFTHENELSE, TLIST has all the diamonds marked on the "T" side of the IFTHENELSE, and FLIST has the diamonds marked on the "F" side. For DOWHILEs, all the diamonds marked are in TLIST. For DOWHILEs FLIST is not used. LASTTJ points to the node of the j of the last embedded diamond on the "T" side of an IFTHENELSE. LASTFJ points to the node of the j of the last embedded diamond on the "F" side of an

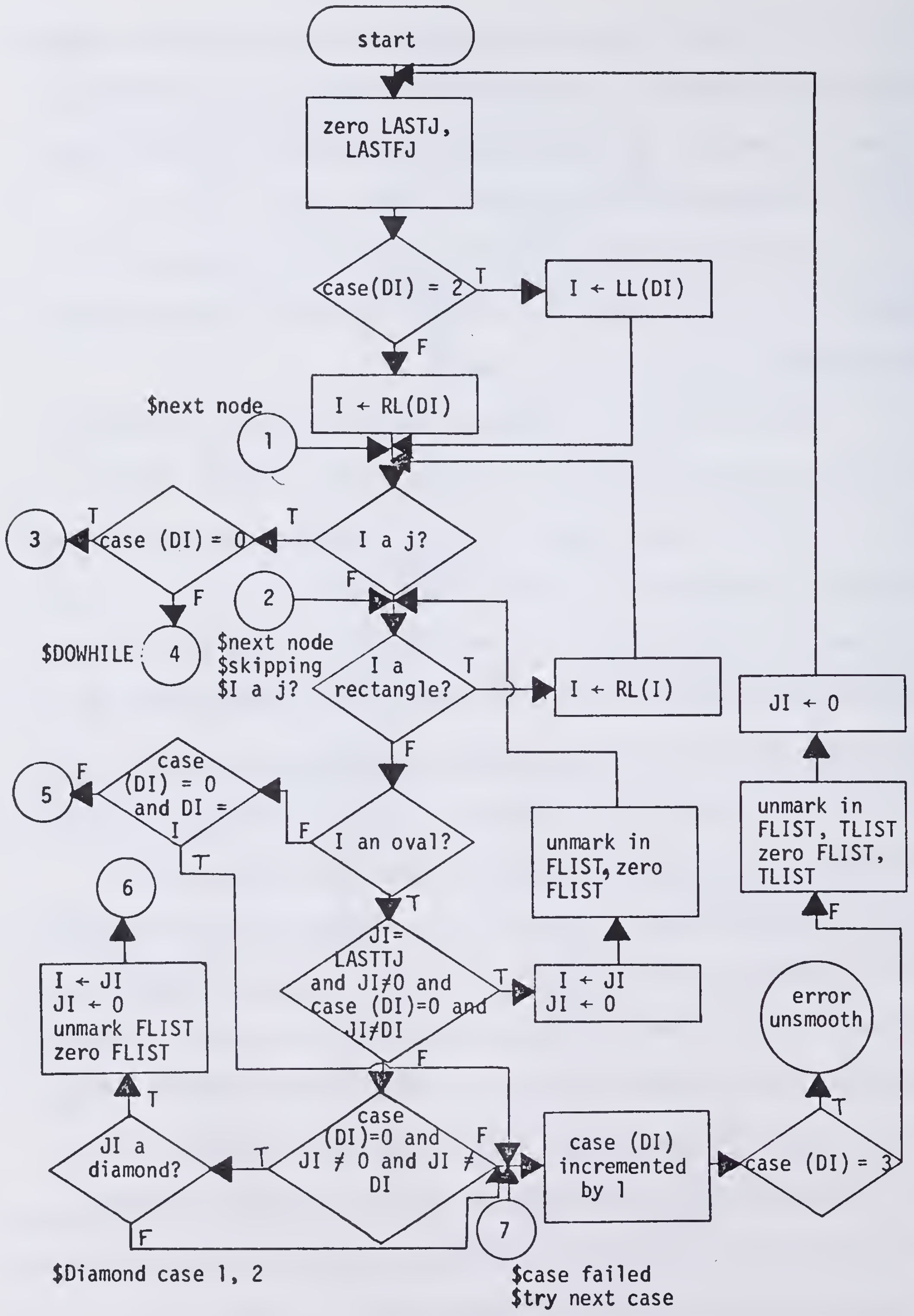


Fig. 5-32. Part I of Flowchart of Routine SR

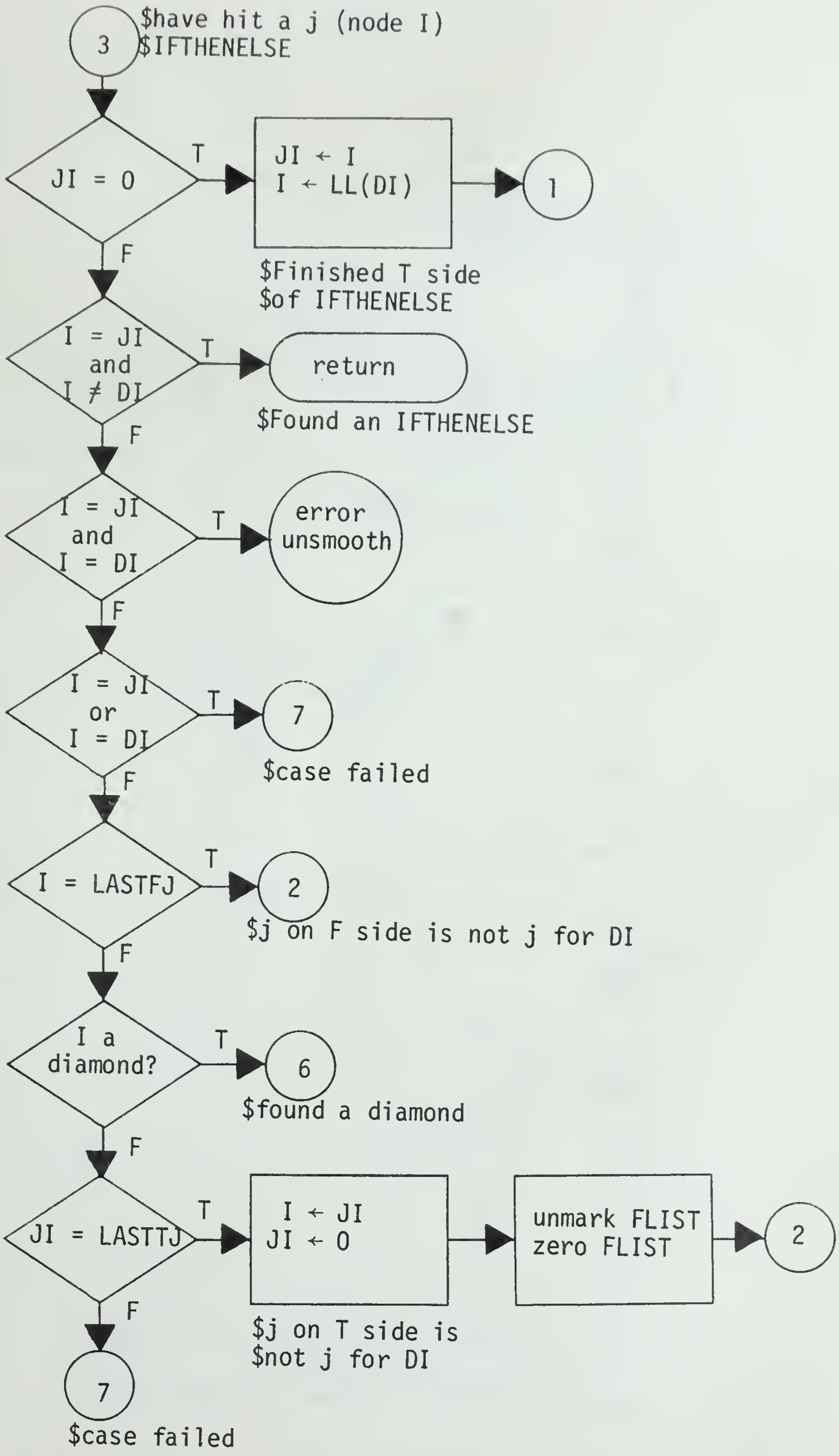


Fig. 5-33. Part II of the Flowchart of Routine SR

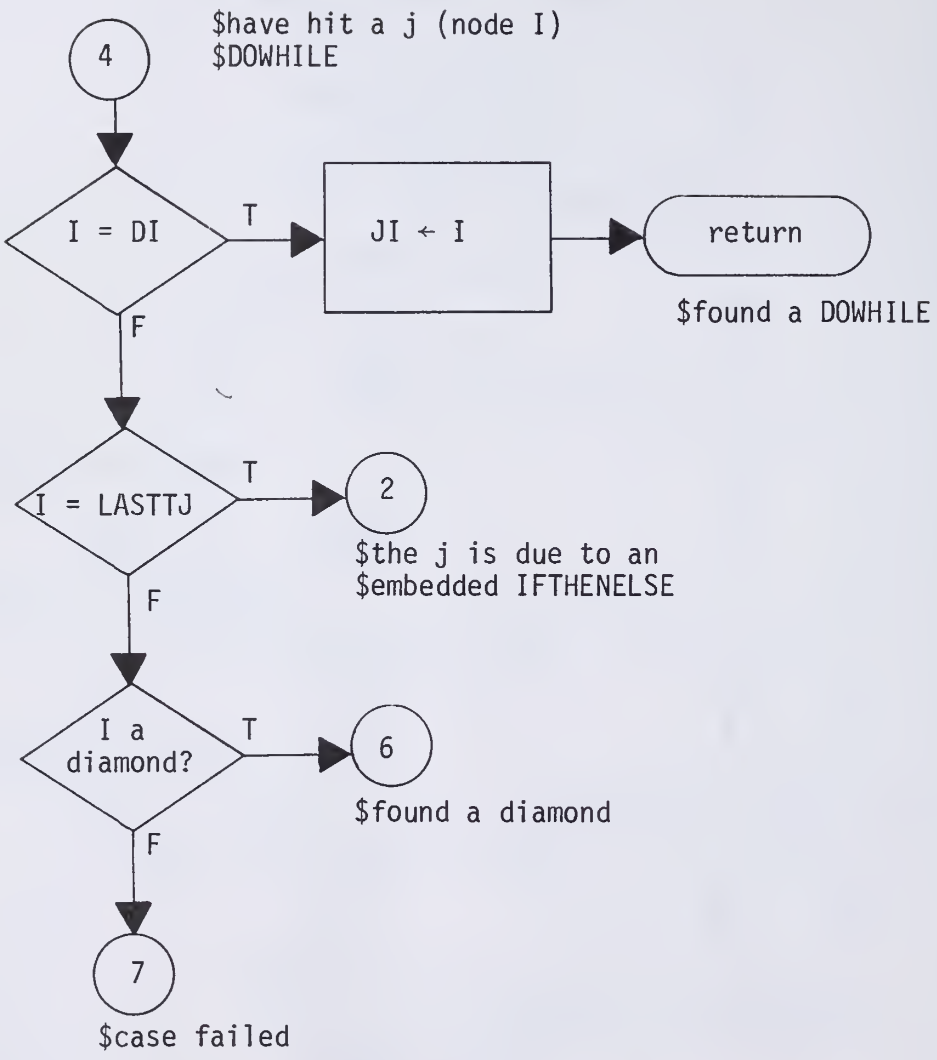


Fig. 5-34. Part III of the Flowchart of Routine SR

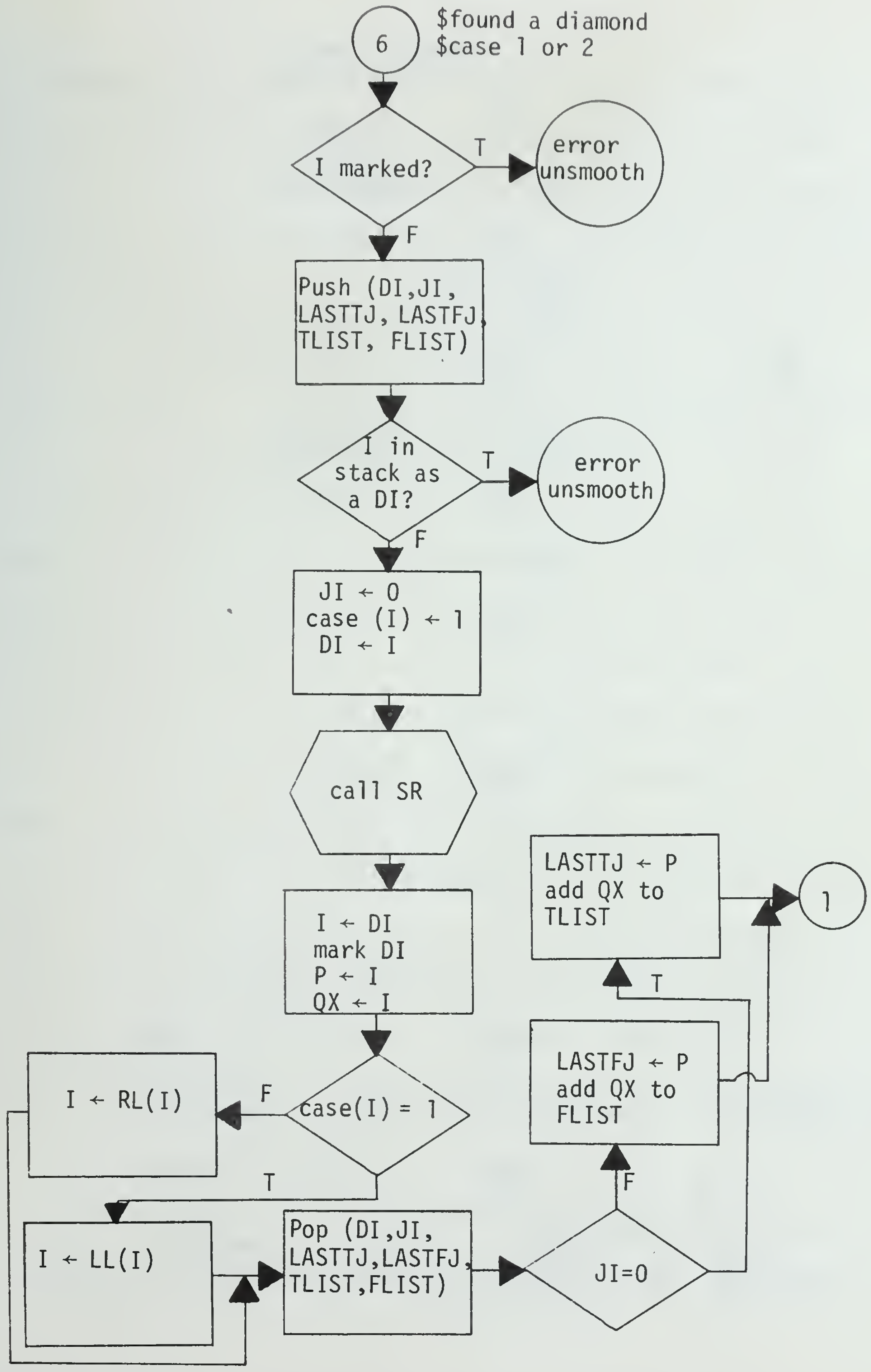


Fig. 5-35. Part IV of the Flowchart of the Routine SR

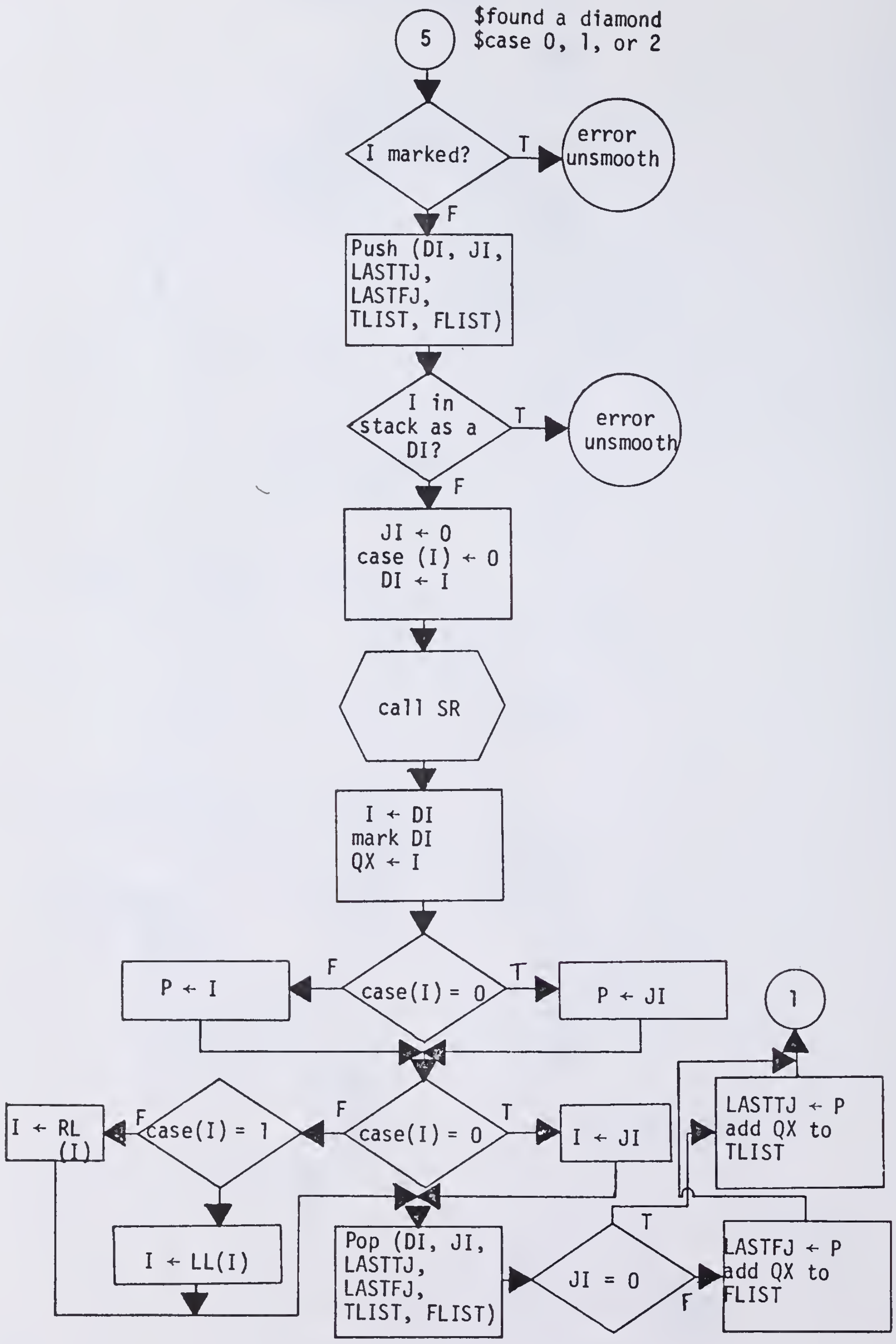


Fig. 5-36. Part V of the Flowchart of the Routine SR

IFTHENELSE. For DOWHILEs, LASTTJ points to the j of the last embedded diamond. For DOWHILEs, LASTFJ is not used. The above pointers DI, JI, LASTTJ, LASTFJ, and the two lists TLIST and FLIST are what is required to be pushed into a stack whenever SR finds a new diamond.

To continue the discussion of Fig. 5-32: routine SR checks whether I is a j. If I is a j, then, depending on the case of DI, SR goes to ③ of Fig. 5-33 or ④ of Fig. 5-34 to see whether SR has found an IFTHENELSE or a DOWHILE respectively. SR's goal is to find a j to finish an IFTHENELSE or a DOWHILE. SR may be detoured by finding a j that is not the correct one for DI. If the node I (cf. Fig. 5-32) is not a j, SR checks whether I is a rectangle. If I is a rectangle, then SR continues with the next node. If I is not a rectangle, SR checks to see whether I is an oval. The discussion of I as an oval is deferred to later. If I is not an oval, it must be a diamond. If the case (DI) = 0 and DI = I, then SR has found a loop. The IFTHENELSE case fails and case (DI) is incremented by one. If the check is false and I is a diamond, SR goes to ⑤ of Fig. 5-36, pushing the current DI into the stack and calling SR to check whether the new diamond is smooth.

<u>IFTHENELSE (case = 0)</u>

In Fig. 5-33, SR assumes that the diamond DI and the found j are part of an IFTHENELSE. Below is shown a flowchart segment with an IFTHENELSE with no embedded diamonds and the list structure representation of the flowchart segment.

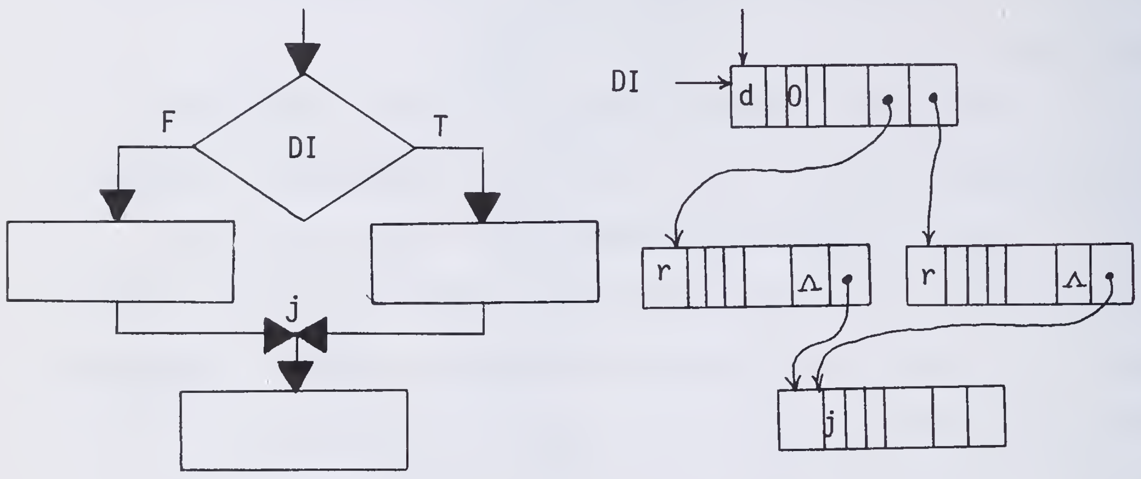


Fig. 5-37. Flowchart Segment of an IFTHENELSE and Its List Structure Representation

With JI initially zero, SR takes the "T" side of the diamond (RL (DI)) and traverses the list structure, moving I until SR finds a j. When I points to j, JI is set to I, the "F" side of the diamond is taken (I ← LL (DI)), and SR tests whether the new node is a j (① Fig. 5-32). When another j is hit and case = 0, SR is again at ③ of Fig. 5-33. Since JI has been set to the j found by traversing the "T" side of the diamond DI, JI is not equal to zero. If I = JI, i.e., the newly found j of the "F" side is the same node as the j of the "T" side, and I ≠ DI, an IFTHENELSE has been found. Routine SR returns to where it was called. DI points to the diamond of the IFTHENELSE, case (DI) is zero, and JI points to the j of the IFTHENELSE.

If I = JI and I = DI, the following situation is true:

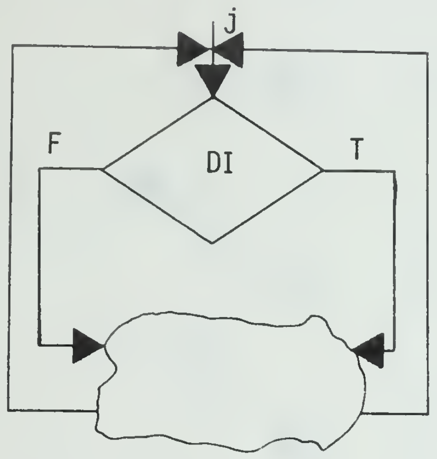


Fig. 5-38. Unsmooth Flowchart Segment When I = JI and I = DI

The flowchart segment above can never be smooth. In the web grammar, one and only one j is allowed for every DOWHILE. Since there are three arrows entering DI, there must be two j's in the web grammar. Therefore, to be smooth, the above must have at least two arrows from an IFTHENELSE. Since two arrows are branching back, they cannot be from an IFTHENELSE. The contradiction proves the above is unsmooth. The student is given a message saying that there is a bad loop with both the "T" side and the "F" side eventually returning to the diamond.

If both (I = JI and I ≠ DI) and (I = JI and I = DI) are false, then this logically implies I ≠ JI. If I does not equal JI, this means that the j on the "T" side and the j on the "F" side are not the same node. Several situations may arise in which the diamond can still be smooth. If I = DI or JI = DI, then SR may have been trying to find an IFTHENELSE which is really a DOWHILE, as shown below.

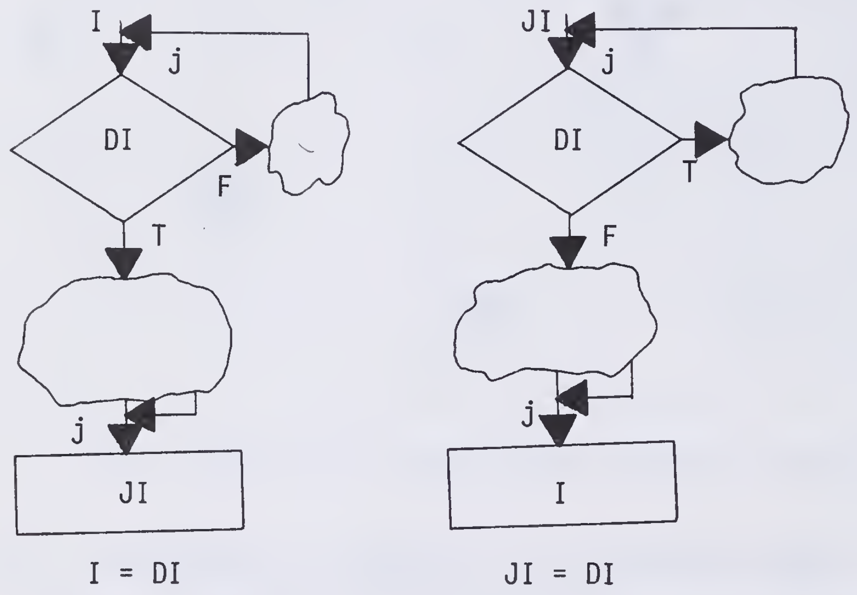


Fig. 5-39. Possible DOWHILEs if I = DI or JI = DI

In the above situations, SR (cf. ⑦ of Fig. 5-32) tries the next case by incrementing the case of DI by one.

If there are embedded DOWHILEs or IFTHENELSEs inside an IFTHENELSE, the condition JI ≠ I could mean that SR found the wrong j for DI. Below is shown a flowchart segment with its list structure which has embedded diamonds in an IFTHENELSE. The nodes have been numbered for reference in the text. It should be noted that

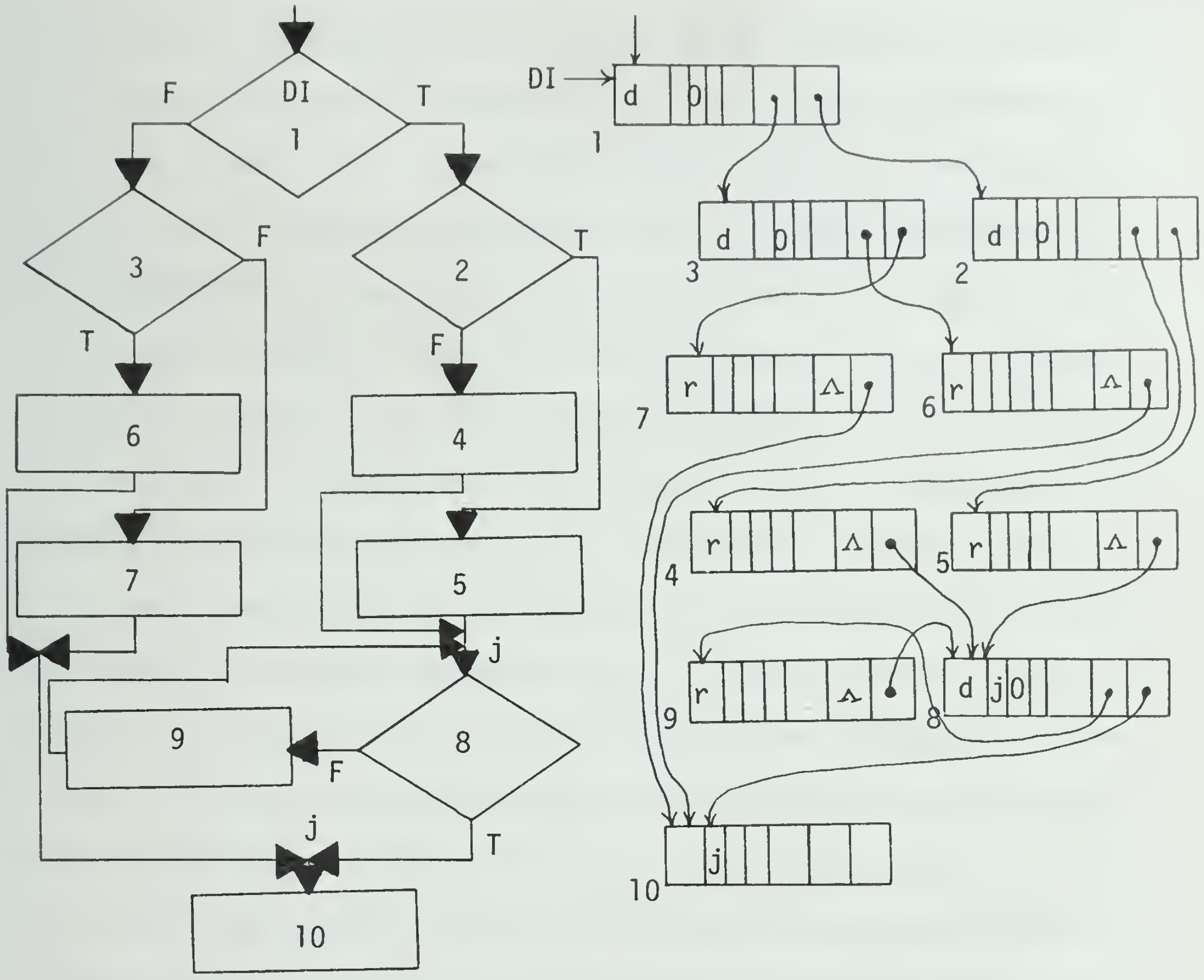


Fig. 5-40. A Flowchart Segment with IFTHENELSE with Embedded Diamonds

there are four diamonds and only two j's in the list structure, and that the above flowchart is smooth.

The analysis is started at the top of Fig. 5-32. The next node is the "T" side of DI (I ← RL(DI)). This I (node numbered 2) is neither a j nor a rectangle nor an oval. I is a diamond and SR goes to ⑤ of Fig. 5-36. The current DI, etc., are pushed and SR is called. For the moment, it will be assumed that SR finds the smaller IFTHENELSE (nodes 2, 4, 5, and 8). I points to node 8 (cf. Fig. 5-36). The stack is popped. DI is again node 1. LASTTJ points to the JI of the smaller IFTHENELSE (node 8). At ① of Fig. 5-32, I is a j and case (DI) is zero. At ③ of Fig. 5-33, since JI = 0, JI is set to I (node 8) and the "F" side of DI is the next node. This node I (node 3) is neither a j nor a rectangle nor an oval. I is a diamond and SR goes to ⑤ of Fig. 5-36. Again, the current DI, etc., are pushed and SR is called. For the moment, it will be assumed that SR finds the smaller IFTHENELSE (nodes 3, 6, 7, and 10). At the bottom of Fig. 5-36, I points to the JI of the smaller IFTHENELSE (node 10). The stack is popped. DI is again node 1. Since JI ≠ 0, LASTFJ points to the JI (node 10) of the smaller IFTHENELSE. SR goes to ① of Fig. 5-32. I (node 10) is a j. SR goes to ③ of Fig. 5-33, since case (DI) = 0. JI ≠ 0 and JI ≠ I. (JI points to node 8.) Since I = LASTFJ, SR goes to ② of Fig. 5-32 and asks whether I is a rectangle. After many computational steps, SR has discovered that the j on the "F" side of the IFTHENELSE is not the same node as the j on the "T" side. The j found by SR on the "F" side had already been used by the smaller IFTHENELSE (I = LASTFJ condition was true). Therefore, SR continues searching for the next j on the "F" side.

SR will continue looking for a j to match JI until SR hits the "stop" oval. In Fig. 5-32, I is tested to see whether it is an oval. If I is an oval, JI = LASTTJ, and case (DI) = 0, and JI $\neq$ DI, then I takes on the value of JI, JI is zeroed, all the diamonds in FLIST are unmarked, and FLIST is zeroed. SR goes to ⓶ of Fig. 5-32, skipping the test "Is I a j?". SR continues down the "T" side of the IFTHENELSE, searching for the next j. The other situation in which an oval may be hit is one in which a j, found on the "T" side of an IFTHENELSE that is a diamond, is not the j of the IFTHENELSE. JI points to this node. While SR is traversing the "F" side of the IFTHENELSE, SR will never find a j equal to JI. Eventually, SR will hit the "stop" oval. In Fig. 5-32, if I is an oval, case (DI) = 0, JI $\neq$ 0, JI $\neq$ DI, and JI is a diamond, then I takes on the value of JI, JI is zeroed, all diamonds in FLIST are unmarked, FLIST is zeroed, and SR goes to ⓺ in Fig. 5-35 to handle the loop. This concludes the details of the way that SR handles an IFTHENELSE. The diagram below summarizes the IFTHENELSE.

SR searches the "T" side of DI until it finds a j. SR then searches the "F" side of DI until it finds a j. If these two j's are the same node, SR has found an IFTHENELSE. If these two j's are not the same node, then the next j on the "F" side is found. If the two j's are not the same, either the next j on the "F" side is found, or SR hits the "stop" oval. If SR hits the "stop" oval, SR searches for the next j on the "T" side and begins searching for the first j on the "F" side. Eventually, the correct j for the

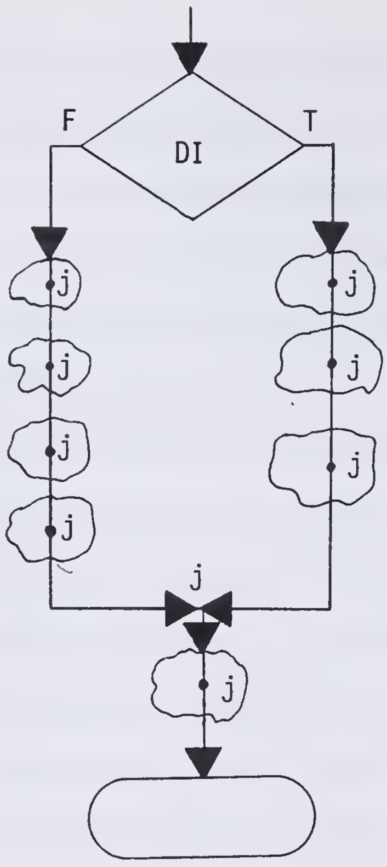


Fig. 5-41. Summary of SR's Search for J's in an IFTHENELSE

IFTHENELSE is found, or all the possibilities are exhausted. If it is the latter, SR tries a DOWHILE by incrementing the case of DI by one. Any time SR finds a diamond, it must push the stack and try to show that the new diamond is part of an IFTHENELSE or a DOWHILE. After succeeding, SR pops the stack and continues. Since every j of an IFTHENELSE may be a j for a later DOWHILE or IFTHEN-ELSE, care must be taken in continuing after an IFTHENELSE.

DOWHILE which Loops on "T" (case = 1)

After SR fails to find an IFTHENELSE, SR tries to find a DOWHILE which loops on "T" (case = 1). SR finds the first j. If this j is DI (cf. Fig. 5-34), SR has found a DOWHILE (case = 1). If it is not the correct j, the j must have been the result of finding a DOWHILE or an IFTHENELSE inside the loop. If the j is not part of an IFTHENELSE (I = LASTTJ is not true) and I is a diamond (cf. Fig. 5-34), then the j may be part of a DOWHILE. If the j is not part of a DOWHILE or an IFTHENELSE, then the diamond DI cannot be part of a DOWHILE which loops on "T." The next and last case is tried (case = 2). Below is a flowchart segment with a DOWHILE which loops on "T." The "a" is assumed to be smooth. If SR hits

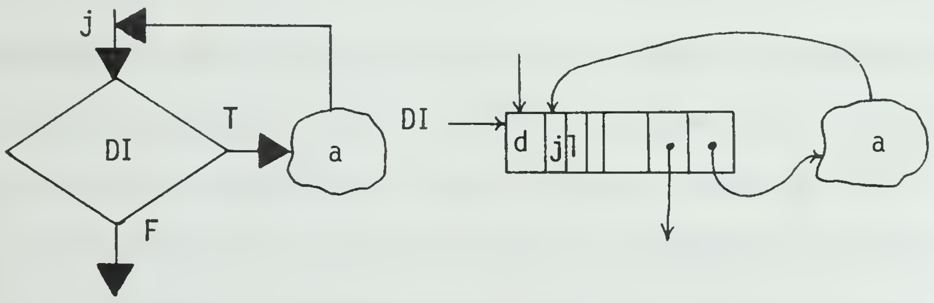


Fig. 5-42. A Flowchart Segment with a DOWHILE which Loops on "T"

an oval, then the diamond cannot be part of a DOWHILE which loops on "T," and SR tries the next case.

DOWHILE which Loops on "F" (case = 2)

If the diamond is not part of an IFTHENELSE or a DOWHILE which loops on "T," the diamond must be a DOWHILE which loops on "F," or the flowchart is unsmooth. For case (DI) = 2 (cf. Fig. 5-32), SR takes the "F" side of the diamond (I ← LL(DI)) and finds the first j. If this j is equal to DI, SR has found a DOWHILE which loops on "F." Below is a DOWHILE which loops on "F" and its list structure. The "b" is assumed to be smooth. If SR hits a j which

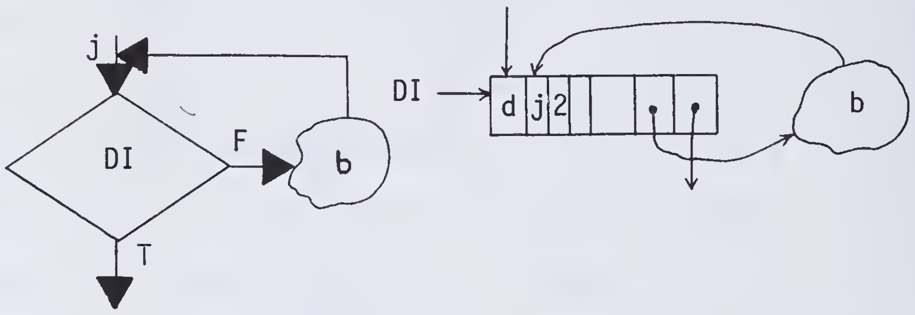


Fig. 5-43. A Flowchart Segment with a DOWHILE which Loops on "F"

is not the correct j for DI, the j must correspond to an IFTHENELSE (I = LASTTJ in Fig. 5-34) or a DOWHILE ("Is I a diamond" in Fig. 5-34) inside the loop. If the j does not correspond to part of an IFTHENELSE or a DOWHILE, SR halts and tells the student that the flowchart is unsmooth and why it is unsmooth. If SR hits an oval during case = 2, the flowchart is also unsmooth.

Embedded Diamonds

In the process of determining the case of a diamond, SR will find other diamonds. Two situations arise: either SR knows that this new diamond must be a loop (cf. Fig. 5-35), or SR has no knowledge of the new diamond and must assume it can be any of the three cases (cf. Fig. 5-36). The only difference between the flowchart segment in Fig. 5-35 and the flowchart segment in Fig. 5-36 is that Fig. 5-35 assumes that the diamond is case 1 or 2 and Fig. 5-36 assumes the diamond can be case 0, 1, or 2. Since Fig. 5-35 and Fig. 5-36 are very similar, only Fig. 5-36 will be discussed.

In Fig. 5-36 if a diamond (I) has been found, SR checks to see whether I has already been marked. If I is marked, then the flowchart is unsmooth. The discussion of why the flowchart is unsmooth will appear in Section 5.4.7. Assuming I is not marked, SR pushes the old DI, JI, LASTTJ, LASTFJ, TLIST, and FLIST into the stack. The newly found diamond I is checked to make sure it is not already in the stack. If I is in the stack, the flowchart is unsmooth. The discussion of why the flowchart is unsmooth will appear in Section 5.4.7, the "Termination of the Smooth Checker." Assuming that I is not in the stack, SR sets the needed initializing conditions ($JI \leftarrow 0$, case $(I) \leftarrow 0$, and $DI \leftarrow I$) and calls SR. If SR is successful in showing that the diamond is smooth, SR returns, I is set to DI, and the diamond DI is marked "finished." A temporary pointer QX points to the diamond that is DI before the pop ($QX \leftarrow I$). QX is added after the pop to TLIST or FLIST, depending on whether JI equals 0. A temporary pointer P points to the j of the diamond that SR found to be smooth. After the pop and depending on whether JI equals 0, LASTTJ or LASTFJ is set to P. If the case of the diamond SR just found to be smooth is zero, I takes on the value of JI. If case (I) = 1, SR takes the nonlooping side ("F" side) of DI. If case (I) = 2, SR takes the nonlooping side ("T" side) of DI. With QX, P, and I set, SR pops DI, JI, LASTTJ, LASTFJ, TLIST, and FLIST off the stack. If JI is not equal to zero, the DI just popped is an IFTHENELSE and SR is on the "F" side of the IFTHENELSE. If JI is not equal to zero, then LASTFJ and FLIST are updated, since SR just found a diamond on the "F" side; otherwise, LASTTJ and TLIST are updated. SR continues at 1 in Fig. 5-32.

5.4.7 Termination of the Smooth Checker

The Smooth Checker must always terminate, whether the student's flowchart is smooth or not. If the Smooth Checker did not terminate, the student would wait forever for PASF to decide whether his/her flowchart is smooth or not. If the student's flowchart is smooth, termination occurs when SR1 finds the "stop" oval. If the student's flowchart is not smooth, SR could search forever if it were not for two checks (cf. Fig. 5-35). Every time SR finds a new diamond, SR checks first that the diamond has not been marked finished and, secondly, that the diamond is not already in the stack. These two checks, which are discussed in detail in this section, guarantee that the Smooth Checker will always terminate.

In Fig. 5-36, whenever SR finds the j for a diamond and returns, SR marks the diamond "finished." If SR finds a diamond that has been already marked "finished," SR halts and gives the message to the student that his/her flowchart is unsmooth. Below is an example in which this check is needed. Without marking each

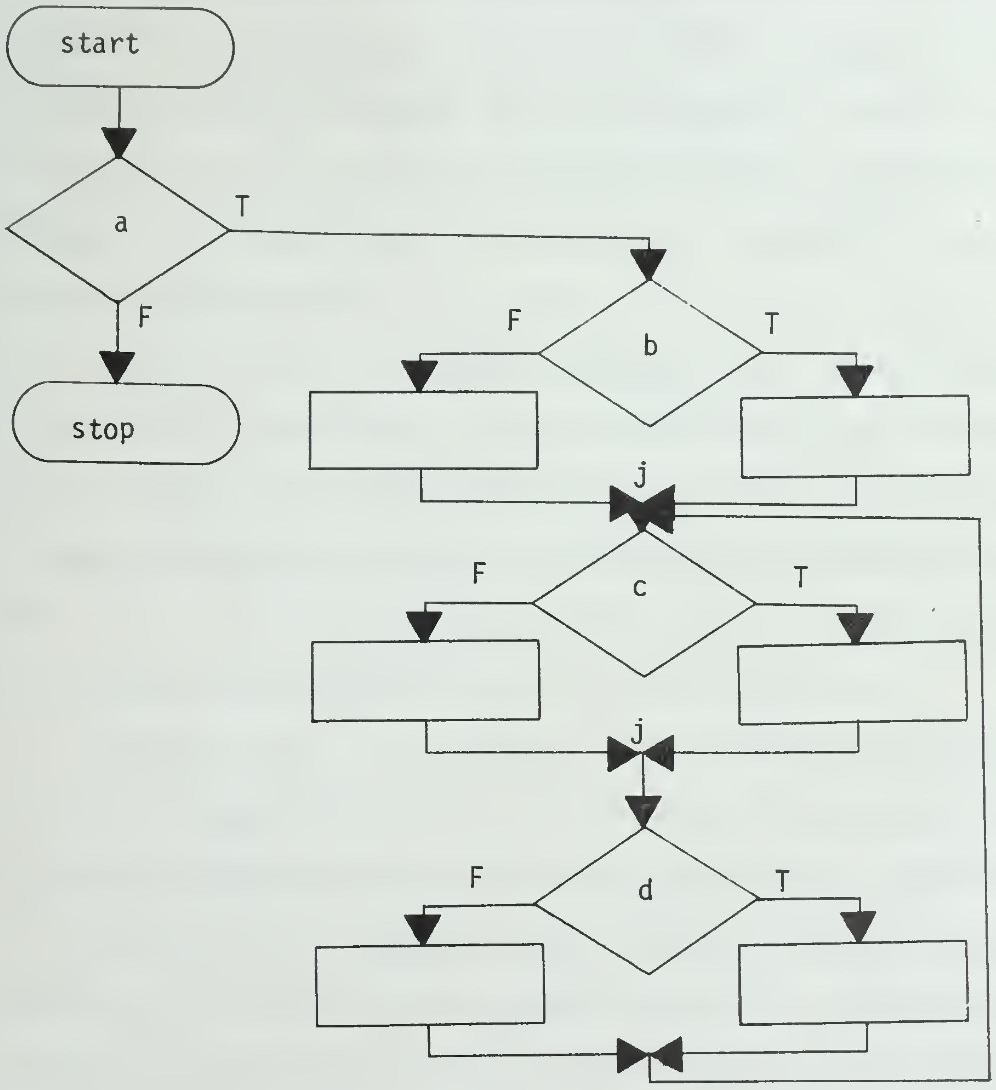


Fig. 5-44. A Flowchart in Which the Mark Check is Needed

"finished" diamond and checking for the mark every time a diamond is hit, the example above would be an infinite sequence of IFTHENELSEs (bcdcdcd•••).

SR takes the "T" side of the diamond "a" and finds a diamond "b." SR finds successfully that "b" is an IFTHENELSE and marks "b" finished. Since "c" is a j, a diamond, and the j could result from the IFTHENELSE of "b," SR assumes "c" can be part of an IFTHENELSE. SR finds successfully that "c" is an IFTHENELSE and marks "c" finished. SR finds "d" is an IFTHENELSE and marks "d" finished. SR hits "c" and finds that "c" is a j and a diamond and the j could result from the IFTHENELSE of "d." SR would again assume "c" is a part of an IFTHENELSE, except that "c" has been marked finished (cf. Fig. 5-36). SR halts and outputs a message to the student saying that his/her flowchart is unsmooth because of a bad loop in a branch of the diamond "a."

In certain situations, SR must unmark some of the diamonds it has marked. Every time SR matches a j for a diamond, SR marks the diamond finished. If SR fails in its attempt to show that a diamond is an IFTHENELSE, SR must unmark all the diamonds it marked trying to show the diamond was part of an IFTHENELSE. For each DI there are two lists kept of the diamonds that have been marked. One list (TLIST) contains all the diamonds marked on the "T" side of an IFTHENELSE. The other list (FLIST) contains all the diamonds marked on the "F" side of an IFTHENELSE. These two lists, together with the corresponding DI, are pushed and popped in the stack. If the j has to be moved on the "T" side, SR restarts the search for the j on the "F" side. When this occurs, all the diamonds in FLIST are unmarked and FLIST is zeroed. When SR is trying to find a DOWHILE, the marked diamonds are placed in the TLIST. Each time SR increments the case, all the diamonds in the TLIST and FLIST are unmarked and the two lists are zeroed. The two lists TLIST and FLIST allow SR to unmark the diamonds necessary when SR backs up in looking for another j in the IFTHENELSE, and when SR tries a new case.

SR progresses through the list structure, systematically marking each diamond node it finishes. Since there are a finite number of diamond nodes, SR will eventually exhaust all the diamond nodes and terminate, unless SR tries to redo an unfinished diamond node. The check of the stack (to be discussed next) guarantees that an unfinished diamond node is not redone.

In Fig. 5-36, if a diamond has been found, the old DI, etc., are pushed into the stack. SR checks to make sure that the newly found diamond (I) is not a DI in the stack. If I is in the stack, the flowchart is unsmooth. If I is in the stack, then SR must have been working on, for example, diamond "a," found another diamond, "b," and pushed "a" into the stack. Before the diamond "b" is shown to be smooth and popped from the stack, SR hits "a" again. Therefore, the flowchart must be unsmooth, because "a"

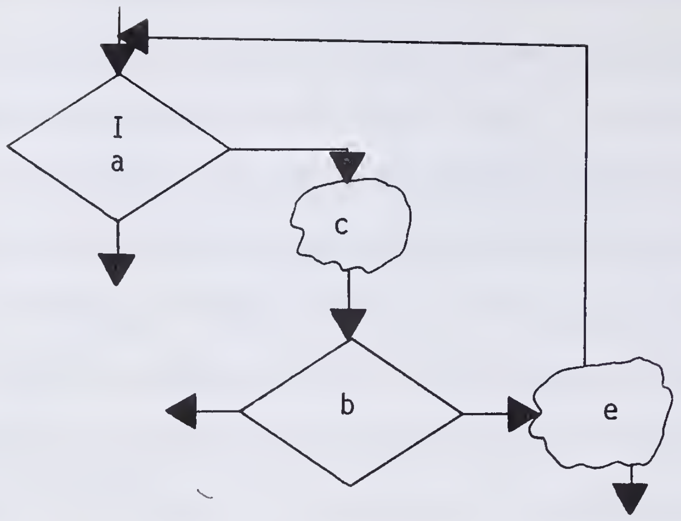


Fig. 5-45. The Situation if I is in the Stack

forms a loop which is not a DOWHILE, since everything inside is not smooth. Below is an example in which this stack check is required.

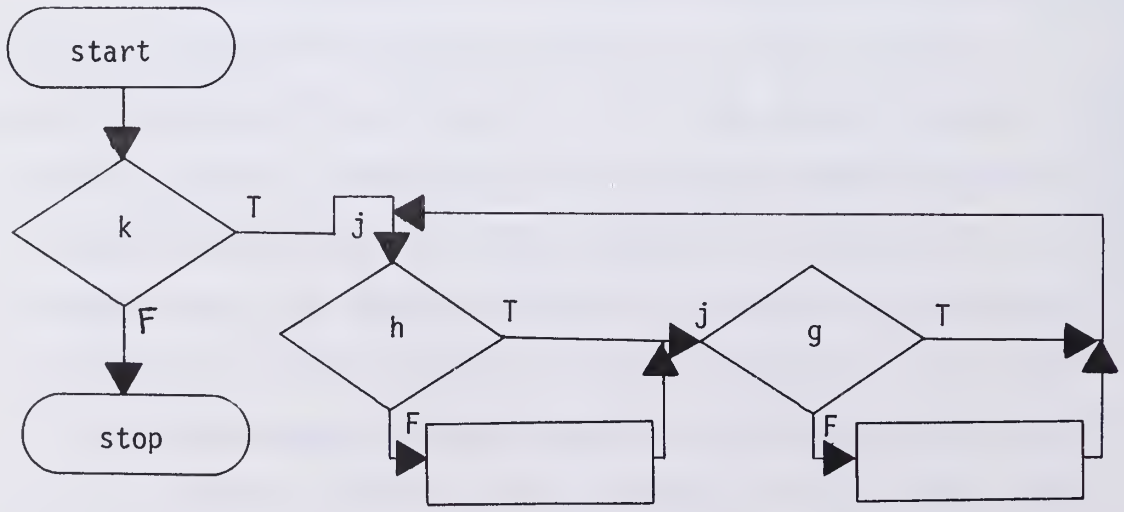


Fig. 5-46. Example in which Stack Check is Needed

Without the stack check, SR would find that the above flowchart is an infinite number of loops (h, g, h, g, h ···). Since "h" is a j, a diamond, and there is no IFTHENELSE in front of "h" to cause a j, SR assumes "h" must be part of a DOWHILE. SR continues on the "T" side of "h" and finds "g," which is also a j and a diamond. SR pushes "h" into the stack. Again, since there is no IFTHENELSE in front of "g" to cause a j, SR assumes "g" must be part of a loop. SR continues on the "T" side of "g" and finds that "h" is a j and a diamond. Again, since there is no IFTHENELSE in front of "h" to cause a j, SR would assume that "h" must be part of a loop, except that "h" is already in the stack. Therefore, the flowchart is unsmooth.

The stack check guarantees that any unfinished diamond is not redone. The stack check, together with the marking of all the finished diamonds, guarantees that the Smooth Checker always terminates.

5.5 Regular Expression Representation Generator

The routine which generates the Regular Expression (RE) representation of the student's smooth flowchart (cf. Section 3.6.4) is a modified version of the Smooth Checker of Section 5.4. The RE generator is simpler, since it does not need to traverse unsmooth flowcharts. The Smooth Checker has set the proper "case" for each diamond (case = 0 is an IFTHENELSE; case = 1 is a DOWHILE which loops on "T"; and case = 2 is a DOWHILE which loops on "F"). The RE generator traverses the smooth flowchart with a recursive algorithm tailored after the smooth checking algorithm. There is no theoretical reason why the RE representation could not be generated during the smooth checking. The division into two tasks was done for the practical reasons of efficiency and easy coding.

As routine SR2 (cf. Fig. 5-47 and Fig. 5-48) traverses the list structure, the contents of each rectangle visited is outputted to an array. If a diamond is found, a "(" is outputted and the routine SRA is called. When SRA returns, SR2 outputs a ")" if an IFTHENELSE (case = 0), or ")*[," contents inside the diamond, and "]" if a DOWHILE (cases 1 and 2). If SR2 hits the "stop" oval, the routine is done. The flowchart of routine SR2 is shown in Fig. 5-47 and Fig. 5-48.

Routine SRA is shown in Fig. 5-49 through Fig. 5-52 with approximately the same format as Fig. 5-32 through Fig. 5-36 for the SR routine of the Smooth Checker. Several portions of SR are missing in SRA: the error portions, the part to try the next case, and the two lists TLIST and FLIST. No marking of the diamonds or checking the stack is necessary for SRA, since the flowchart is known to be smooth. Since the case of DI is known, Fig. 5-35 and Fig 5-36 of SR have been combined into Fig. 5-52 of routine SRA. SRA has extra portions to output the RE representation into the array.

To output the RE representation in the correct order, SRA places the partial strings into a BUFFER, then moves the strings in the BUFFER to the array at the appropriate moment. If

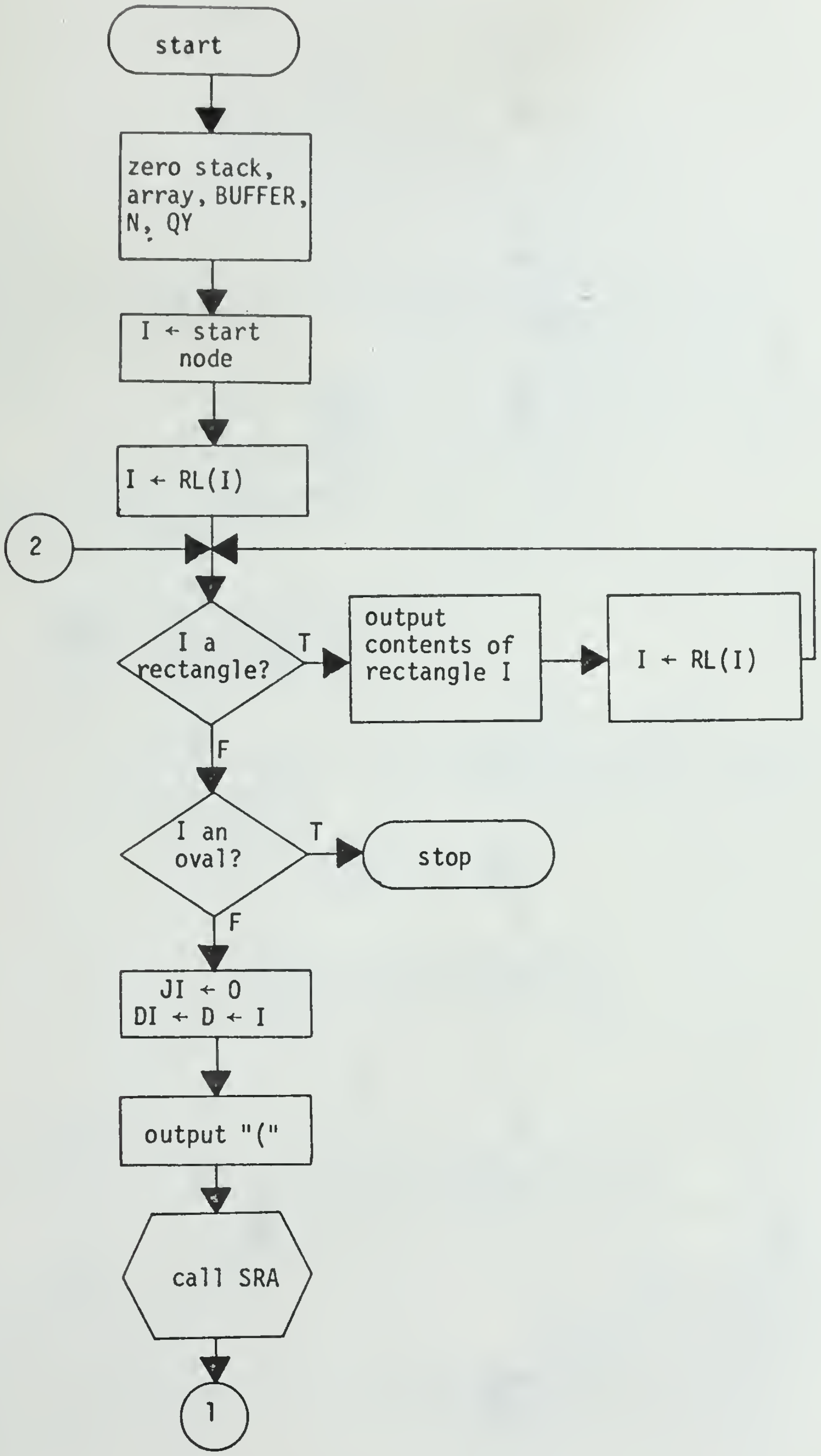


Fig. 5-47. Part I of Flowchart of Routine SR2

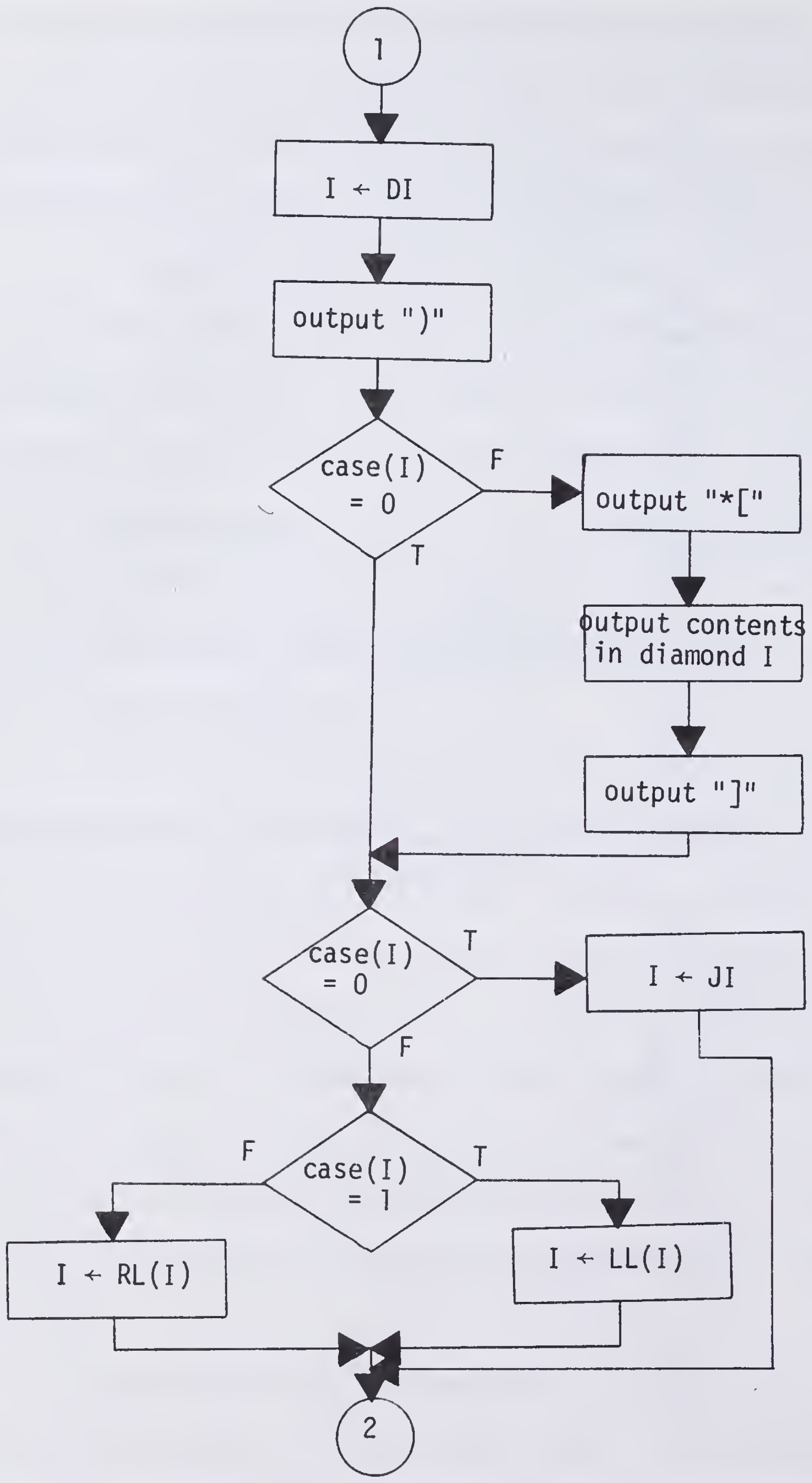


Fig. 5-48. Part II of Flowchart for Routine SR2

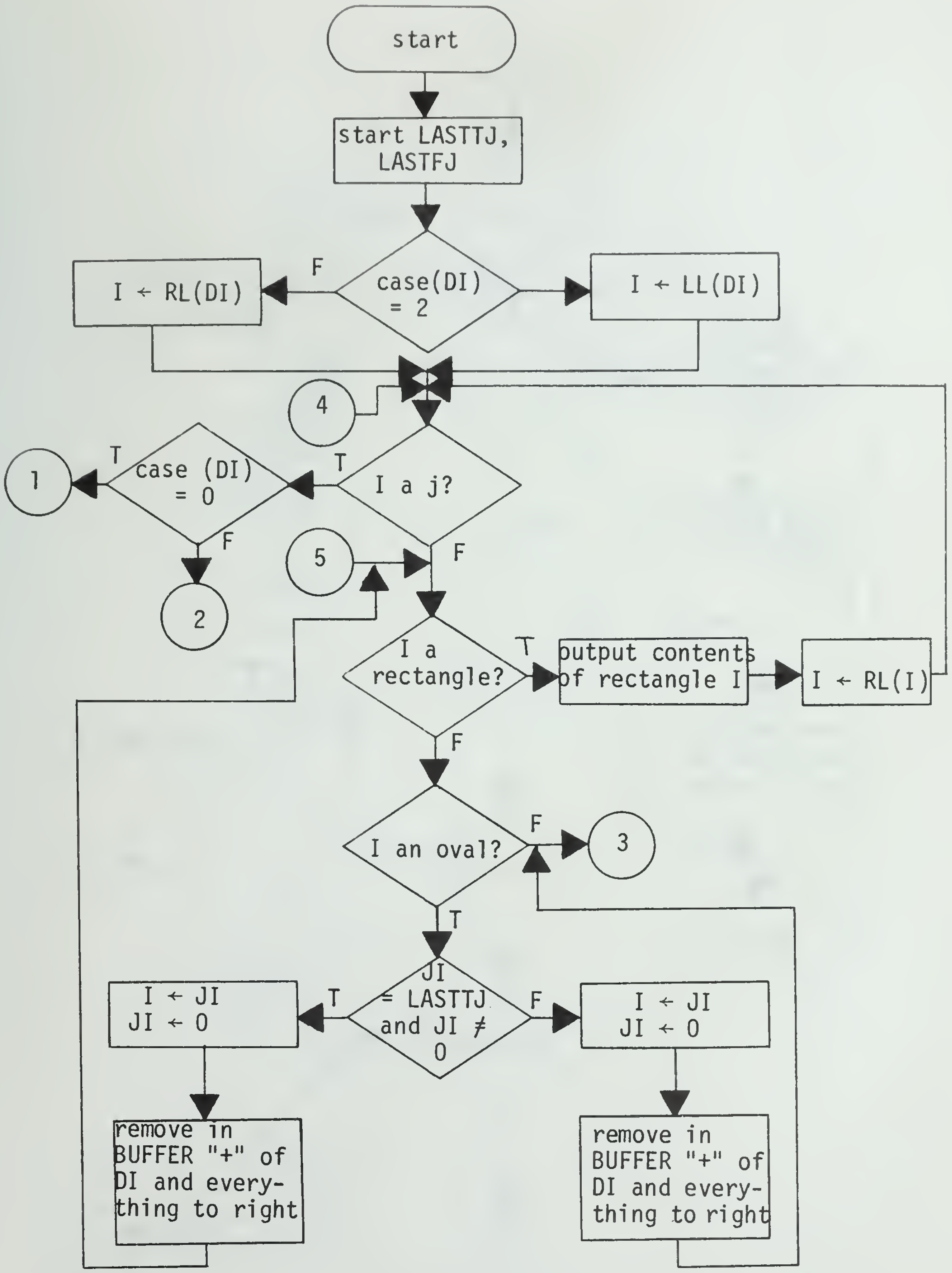


Fig. 5-49. Part I of Flowchart of Routine SRA

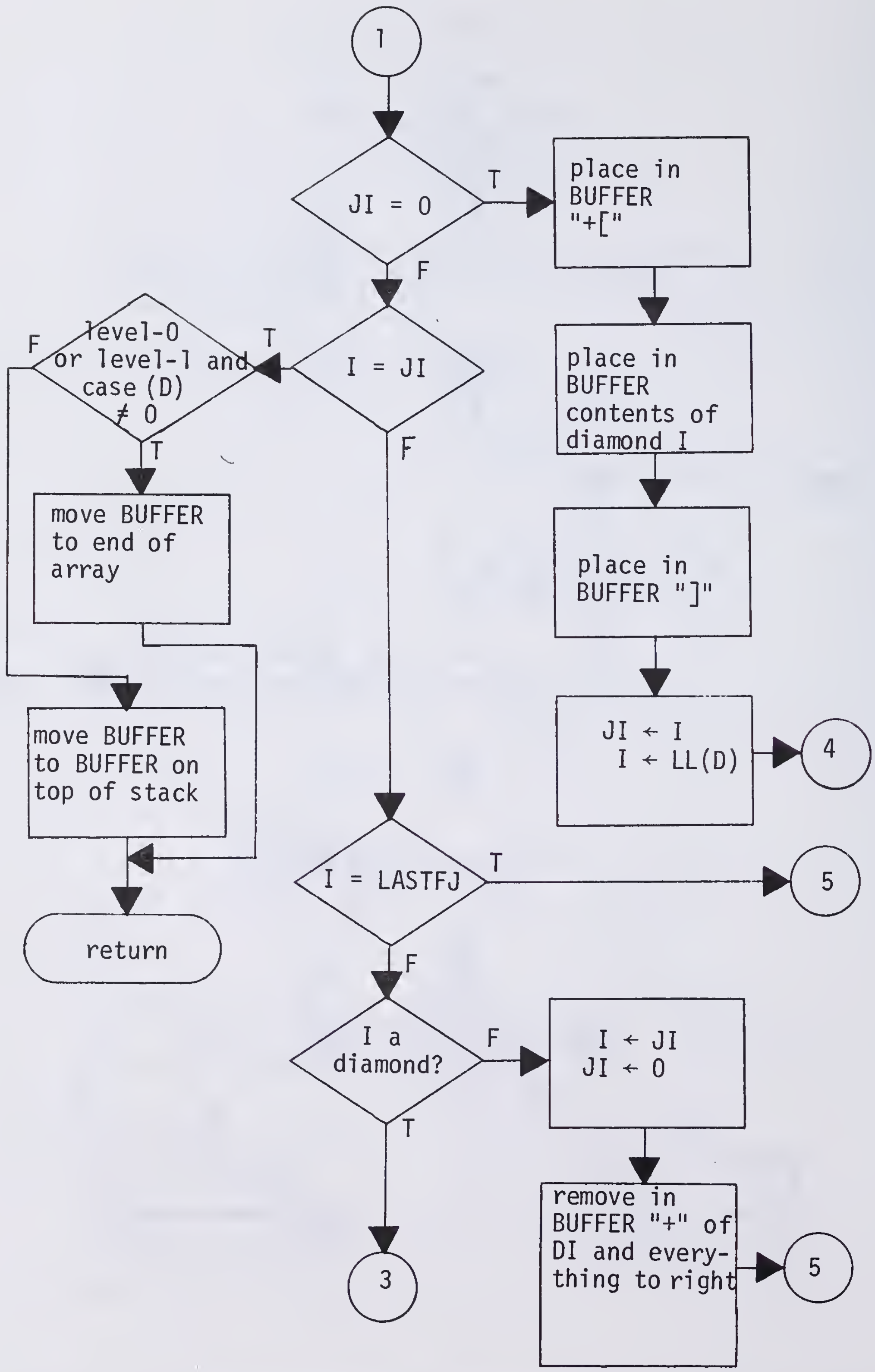

1
JI = 0
T
place in BUFFER "+["
F
place in BUFFER contents of diamond I
I = JI
T
F
level-0 or level-1 and case (D) ≠ 0
F
T
move BUFFER to end of array
place in BUFFER "]"
move BUFFER to BUFFER on top of stack
JI ← I
I ← LL(D)
4
return
I = LASTFJ
T
5
F
I a diamond?
F
I ← JI
JI ← 0
T
remove in BUFFER "+" of DI and every-thing to right
3
5

Fig. 5-50. Part II of Flowchart of Routine SRA

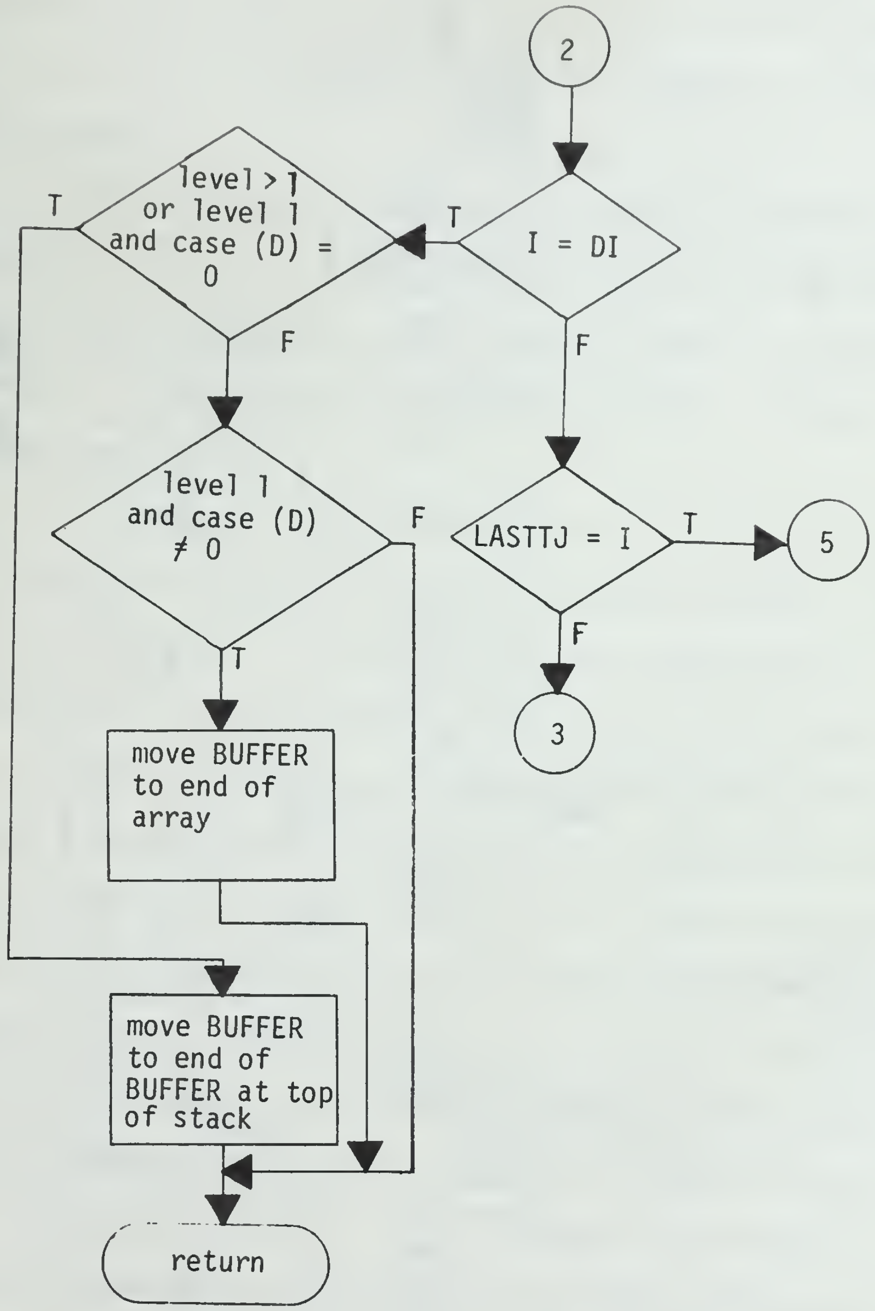

2
I = DI
T
F
level > 1 or level 1 and case (D) = 0
T
F
level 1 and case (D) ≠ 0
F
T
LASTTJ = I
T
F
5
3
move BUFFER to end of array
move BUFFER to end of BUFFER at top of stack
return

Fig. 5-51. Part III of Flowchart of Routine SRA

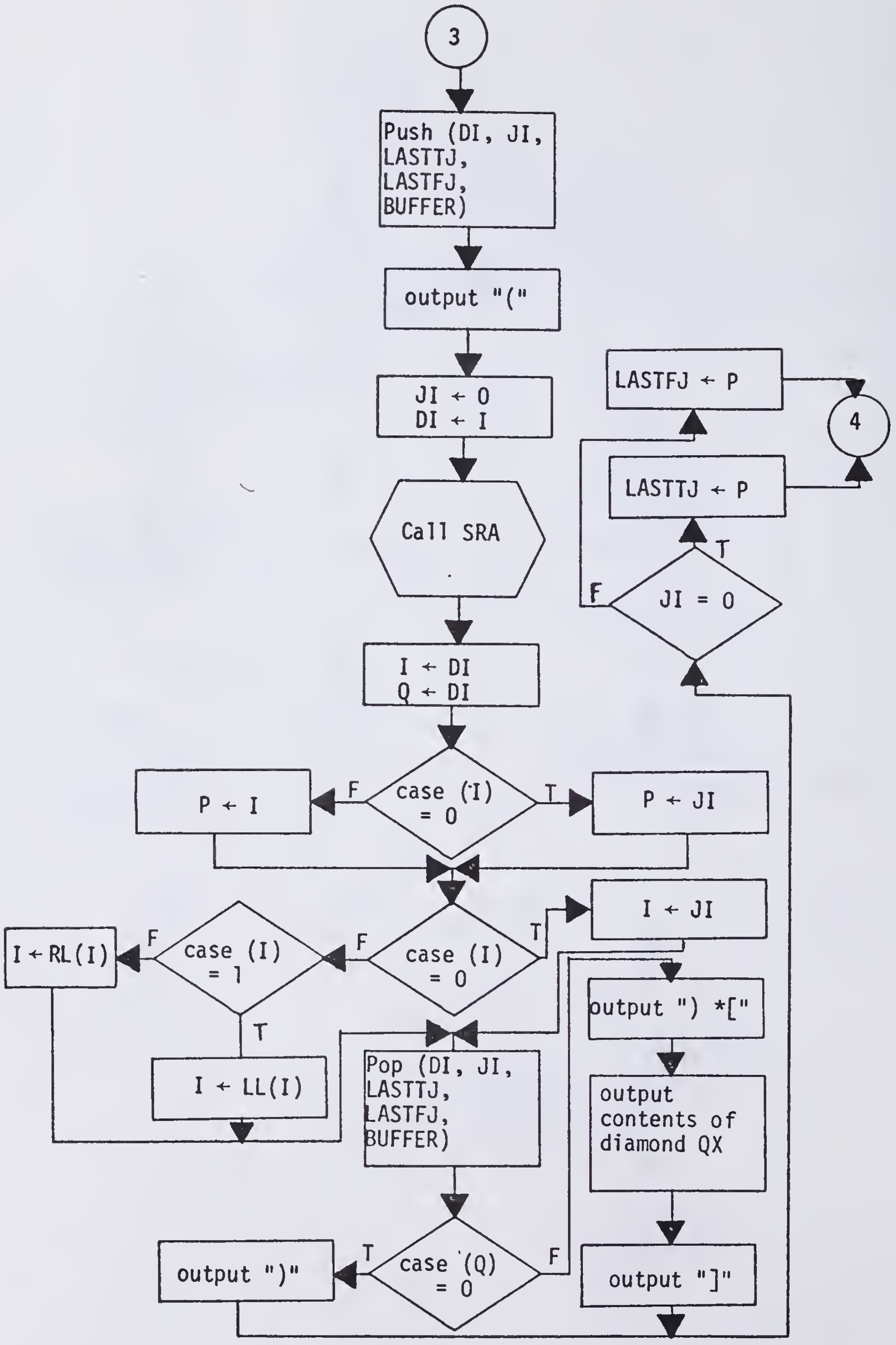


Fig. 5-52. Part IV of Flowchart of Routine SRA

the diamond is a DOWHILE at level 0, SRA does not need the BUFFER and outputs the symbols directly to the array. For other instances, i.e., case = 0 or level greater than 0, the symbols are placed into the BUFFER.

If SRA finds a diamond, SRA outputs a "(," pushes DI, JI, LASTTJ, LASTFJ, and BUFFER into the stack, and calls SRA. Upon returning, SRA outputs a ")." If the diamond is a DOWHILE, SRA outputs "*[," contents of the diamond, and "]." SRA pops the stack and continues.

If a diamond is an IFTHENELSE (case = 0), SRA (cf. Fig. 5-50) finds the first j on the "T" side of the diamond DI. At this point, SRA places "+[," contents of the diamond, and "]" in the BUFFER. SRA continues taking the "F" side of the diamond DI (I ← LL(DI)). If SRA finds an oval or a j on the "F" side of DI which is not the same node as JI, SRA must erase from the BUFFER the "+" of the DI and everything after the "+." This allows SRA to back up and try the next j on the "T" side of DI. SRA does not remove "+," etc., from BUFFER if the j on the "F" side could be the j of an IFTHENELSE just "before" (I = LASTFJ in Fig. 5-50) or the j of a DOWHILE (I is a diamond).

When SRA finds the j of an IFTHENELSE, SRA either concatenates the contents of the BUFFER to the end of the array or concatenates the contents of the BUFFER (level N) to the end of the BUFFER at the top of the stack (level N-1). SRA does the former concatenation if level-0, or if level-1 and the level-0 diamond is not an IFTHENELSE. If SRA hits a diamond which is a DOWHILE (case = 1 or case = 2), SRA searches for the j for the DOWHILE. If SRA finds a DOWHILE, SRA either concatenates the contents of the BUFFER to the end of the array or concatenates the contents of the BUFFER (level N) to the end of the BUFFER at the top of the stack (level N-1). SRA does the former concatenation if level 1 and the level-0 diamond is not an IFTHENELSE.

When routine SR2 finds the "stop" oval, the RE representation is in the array and complete except for one minor detail. For all the diamonds that are a DOWHILE which loops on "F" (case = 2), the condition inside the diamond is negated in the array (e.g., "$a > 10$" would become "$a \leq 10$").

5.6 Algorithm for Translator Check

As part of the deduction scheme (cf. Section 3.7), the two translators TRANSLATOR B and TRANSLATOR R interchange statement types (e.g., add Oab:c;), P's, or strings of statement types and P's (cf. Section 3.7.7). This interchange is only possible if the three conditions of Section 3.7.7 are satisfied. This section will discuss the algorithm which efficiently implements these three conditions in PASF.

The general case is AB interchanged to BA where A and B are strings of "items." An item is either

1. A statement type with its sequence number, list of input variables and constants, list of output variables, and list of maybe-output variables, or

2. A P with its sequence number, list of input variables and constants, list of output variables, and list of maybe-output variables.

Below are examples of items.

```
P3 elf:c;d
sub 0 mn:z;
```

Fig. 5-53. Examples of Items to be Interchanged

The three conditions of Section 3.7.7 need four entities:

1. The input variables of A,
2. The output and maybe-output variables of A,
3. The input variables of B, and
4. The output and maybe-output variables of B.

Since the number of variables that can be used by the student is less than 60, a computer word (60 bits longs) is used for each of these four entities. These four computer words are called IV(A), OV(A), IV(B), and OV(B) respectively. When an input variable (e.g., its name is ninth in the symbol table) is found in A, the corresponding bit (ninth bit) is set to one in IV(A). The three other computer words are similarly used. On one pass of the string AB, the four words can be completely set. To test the three conditions (cf. Section 3.7.7), the following IF statement is performed.

```
IF(IV(A)MASK OV(B)=0)AND(IV(B) MASK OV(A)=0)AND(OV(A) MASK OV(B)=0)
THEN OK;ELSE FAIL;
```

The MASK function is a bit-wise "and" of the two computer words.

If computer word x and computer word y have no bits in corresponding positions set to one, then x MASK y is equal to 0. The above test is performed very quickly by the machine.

The test to allow the interchange AB to BA is quickly performed. The test requires one pass of the string AB and a single computation of a boolean expression.

Chapter Six

CONCLUSIONS AND FUTURE RESEARCH

6.1 Conclusions

The feasibility of the Semantic Formulation Method (SFM) to grade students' programs by machine has been demonstrated in this thesis. SFM is an efficient approach which shows that a student's structured program (i.e., a limited control structure with DOWHILEs and IFTHENELSEs) is correct. To show that the student's program is correct, SFM demonstrates that the student's program is Global Semantic Equivalent (GSE) (cf. Section 2.4) to the instructor's correct answer. Global Semantic Equivalence is computational equivalence with local syntactic transformations which are semantically equal, and with interchanging of independent processes. The feasibility of SFM is proven in the previously described program PASF. PASF, a working program on the PLATO IV system that implements in a rudimentary way the notions of SFM, has been designed to operate in the computer-based educational environment of PLATO IV. SFM is especially well suited for the computer-based educational environment because of its efficiency in the use of storage and computation and its ability to interact with students concerning their programming errors. SFM has been shown to be a viable method of grading student generated structured programs in the PLATO IV computer-based education environment.

PASF's overall goal is to teach the step-wise refinement technique of structured programming to students in an introductory computer science course. The teaching of this technique is improved if the programming language used by the students allows only a limited control structure which involves only DOWHILEs and IFTHENELSEs. Therefore, PASF is an ideal application for SFM which requires structured programs using DOWHILEs and IFTHENELSEs. Since PASF teaches introductory nonmajor students in the first or second week of a 100-level course, it utilizes flowcharts. The constraints of using structured programs and using flowcharts evolved a class of flowcharts called smooth flowcharts. In the process of teaching the step-wise refinement technique, PASF grades smooth flowcharts by the efficient SFM method.

The value of SFM lies in its ability to represent a student's structured program as a string (Regular Expression representation, cf. Section 3.6.4). Representing a program as a string allows magnitudes of speed-up in the required computation, as compared with other representations, e.g., a list structure representation. Representing a problem as a string to obtain efficiency, and at the same time possibly destroying the structural properties, was a common pitfall of Artificial Intelligence (AI) programs of the early Sixties. McCarty [McCarty, 1971] stated that representations that eliminated the inherent structure of the problem was the chief reason for the failure of GPS [Ernst and Newell, 1969] and other

AI progams. In SFM's case, however, the string representation has not destroyed the inherent structure of the student's program. Enough information is in the string (RE representation) to reconstruct the student's flowchart if necessary, since no structural information is lost. Further, the string representation is efficiently stored. Much of the efficiency of SFM in both storage and speed of computation is accredited to the fact that the student's program is represented as a string.

Not impressed by the results of the "syntax-semantic" paradigm of the Sixties, the author decided to incorporate the "description, representation, and deduction" paradigm [Minsky, 1972] in this thesis. Surely, any program which deals with programming languages must worry about syntax and semantics. The author does not advocate the elimination of the use of syntax or semantics in an AI program such as PASF; he only suggests that the program not be designed around the "do all the syntax"-then-"do all the semantics"-model. The design of PASF follows the "description, representation, and deduction" paradigm currently popular in the AI field.

Using this paradigm as a framework for PASF, the <u>description</u> of the programming domain in PASF consists of the process boxes with simple assignment, input, and output statements and the control structure smooth flowcharts to connect the process boxes. A large class of programs can be written in this programming domain.

A large portion of the possible description of a programming domain has been purposely neglected: missing from the description are different data types and data structures. Since PASF deals with students in an introductory computer science course, different data types and data structures were eliminated for pedagogic reasons. The description of possible programs in PASF consists of process boxes in a smooth flowchart.

Each of the three representations of a program utilized in three different portions of PASF is well suited for the necessary manipulations in its portion of PASF. No one representation for all three portions of PASF would be as efficient as each representation is for its own portion. The list structure representation is well tailored to the adding and deleting of boxes and arrows of the input section. The heuristic and algorithm-specific checks are quickly performed on the RE representation. The deduction scheme runs efficiently on the Standard Regular Expression (SRE) representation. Converting from one representation to another might be expensive in terms of computation. In PASF, converting costs little, since only two conversions are needed each time a student checks his/her flowchart. In the time (less than two seconds) the student waits for PASF to check whether his/her flowchart is smooth, PASF also generates the RE representation from the list structure representation.[15] The other conversion which is done

[15]This demonstrates a design philosophy used throughout PASF of distributing all possible computations between student interactions.

at the beginning of the deduction scheme is quick, since the two representations (RE and SRE) are relatively similar. The flexibility of three different representations of a student's program allows PASF to perform a diverse set of tasks quickly and efficiently.

A representation should be as formal and as structured as possible to allow deductions to be performed efficiently, as well as be general enough to cover a useful description. This is the AI researcher's dilemma, restated by Minsky, "We want. . .for the smallest initial structure, the greatest complexity."[16] A broad description will require a complex representation; in that case, the deduction may be formidable or impossible. The trend in AI is to narrow the problem domain by limiting the description (e.g., Winograd's block world [Winograd, 1971]). The programming domain of PASF is vastly limited (e.g., no procedure calls or pointer variables) for this reason. Limiting the problem domain is not enough to guarantee that a deduction scheme will work; an AI program requires "good" representations. In this context, the only known way to define a "good" representation is to say that it is "one that works." The search for "good" representations is a major research effort in the AI field today (especially for "good" representations of knowledge [Winograd, 1974]). For PASF's deduction

[16]Minsky, Marvin, editor, Semantic Information Processing, The MIT Press, Cambridge, Mass., 1968, p. 12.

scheme, the SRE representation (cf. Section 3.7) has been shown to be "good" in the sense discussed above.

The <u>deduction</u> scheme of PASF is specifically tailored to deduce that two programs are equivalent. For reasons of efficiency, the deduction scheme is not a general scheme (e.g., a scheme over first order predicate calculus), but a special purpose scheme to handle only pieces of a structured program. The design of PASF's deduction scheme is modeled after the general theorem provers based on the resolution principle [Robinson, 1965; Henschen, 1971]. Many portions of PASF's deduction scheme are analogous to portions of a general theorem prover. PASF's deduction scheme consists of a SELECTOR, MATCHOR, REDUCOR, and others (cf. Fig. 3-11). The SELECTOR together with the MATCHOR builds a Semantic Model. The Semantic Model consists of pieces (P's) of the student's program which are individually semantically equivalent to pieces (P's) of the instructor's program. The set of a P in the student's program and the corresponding semantically equivalent P in the instructor's program is analogous to an axiom in the general theorem prover.

{Pk of student is semantically equivalent to Pk of instructor} $\rightarrow$ T

{Axiom A_1} $\rightarrow$ T

Fig. 6-1. Analogy of P's of PASF Deduction Scheme and Axioms of Theorem Prover

Each set in the above figure implies Truth. The REDUCOR searches for inconsistencies in the Semantic Model. The REDUCOR reduces two P's in the student's program and two P's in the instructor's program to one P in each. To continue the analogy: a reduction step is analogous to resolving two axioms. Similar to the theorem prover which can resolve two resolved axioms (clauses), the REDUCOR can reduce two reduced P's. In a theorem prover, selecting the next two clauses to resolve requires strategies, e.g., unit preference. Likewise, the REDUCOR has different possible ways to reduce two P's and needs a strategy to choose which two P's to reduce next. If a theorem prover can deduce an empty clause, the theorem has been proved. Similarly, if the REDUCOR can reduce the two SREs to a single P in each, then the student's program has been shown to be correct. The analogy, although not perfect overall and weak in many places, is a useful tool for pointing out areas of future research on PASF-like systems. The deduction scheme of PASF deduces that two programs are equivalent by forming a Semantic Model and reducing the Semantic Model to two single P's.

In the process of reducing to a single P, the deduction scheme checks for consistency between pieces of the student's and the instructor's programs. As the REDUCOR reduces two P's, the variables of the student and the instructor are dynamically bound and checked for use before being given a value. In case of an inconsistency, the REDUCOR tries to move processes in the student's program around to achieve a closer fit to the instructor's program. The REDUCOR via TRANSLATOR R may succeed in moving any independent process of arbitrary size, e.g., two loops.

The whole deduction scheme of PASF can be viewed as a heuristic search through a state space (for an excellent discussion of heuristic searches through state spaces, see [Nilsson, 1971]). The SELECTOR chooses a path through this state space, forming a Semantic Model. When the REDUCOR shows that the Semantic Model has an inconsistency, the deduction scheme backtracks and the SELECTOR chooses a different branch in the state space. When the SELECTOR exhausts all paths through the state space, the deduction scheme halts, giving error diagnostics to the student. In any AI program, the size of this state space is critical: if it is too large, combinatorial explosion forces the search of the space to go for hours; if it is too little, the program is not very intelligent. PASF, to rid the state space of unprofitable branches (this process is called tree pruning), executes the heuristic checks and the algorithm-specific checks. By this pruning PASF is assured that the student's program is reasonably close to the instructor's program. By tuning the SELECTOR, the state space can be enlarged if it is too small. Since searching this state space requires a large share of the CPU time, the size of the state space is closely related to the length of time that the student must wait for PASF to grade his/her flowchart.

The SFM method as demonstrated by PASF is a useful technique for grading students' structured programs. The SFM method should <u>not</u> be used for program verification, which is somewhat related to grading, but different from it: grading involves comparing the student's program with the correct answer and giving feedback on the errors; program verification involves proving a program is correct for the program's specifications. Although grading and program verification both want the program to be equivalent to the correct entity, the equivalences are of a different type. Grading results in Global Semantic Equivalence (close to computational equivalence), while program verification results in input/output equivalence. PASF-like graders which can say that a student's program is correct will become more important in the future as more and more students are enrolled in introductory computer science courses.

6.2 Future Research

As noted above, there is an increasing need for PASF-like graders. This need is especially acute in the teaching of programming in the computer-based education environment. Naively, one may think that using a computer-based education system to teach programming should be easy, i.e., "a computer to simulate a computer." In reality, teaching programming via computer-based education is as hard as or harder than teaching other subjects, e.g., French or history. "A computer to simulate a computer programmer" is a better metaphor (informally proposed by Dr. Donald Gillies at the University of Illinois). Since one activity of many computer programmers is grading student programs, an area of research that needs further investigation is describing and modeling the processes involved in a human grader.

PASF has been attempting to incorporate at a gross level some of the processes of one human grader. The heuristic checks and algorithm-specific checks were derived from the author's experience in grading student programs. PASF is only a start in the right direction: it can be extended and improved within the framework of the SFM method described above. An easy improvement is to include more powerful heuristic checks or algorithm-specific checks, e.g., run the program with test data or find and compare all the traces through the two programs. Data types and data structures, e.g., arrays, should be investigated. The SELECTOR can be made more intelligent. Currently, the SELECTOR is statement or box-oriented: its orientation could be groups of statements instead. The handling of DOWHILEs and IFTHENELSEs by the deduction scheme could be made more general.

Many features of PASF can be extended, but most require more storage and computational time. PASF is currently near the saturation point in storage and computation time for the PLATO IV system. Moving the SFM method out of the computer-based education environment into a purely research environment would open the doors to the investigation of many research problems. For an example, one research problem is implementing the SFM method or a descendant version of SFM in a larger programming domain such as a subset of PASCAL.

BIBLIOGRAPHY

Adams, J. M., "Teaching Declarative Programming," SIGCSE Bulletin, Feb. 1975, Vol. 7, No. 1, pp 83-85.

Aho, A. V., "Design of Efficient Algorithms," Department of Computer Science Colloquium, University of Illinois, Nov. 21, 1974.

Aho, A. V. and Ullman, J. D., "Transformations on Straight Line Programs," Conf. Record Second Annual ACM Symposium on Theory of Computing, pp 136-148 (May, 1970).

Allen, F. E., "Program Optimization," Annual Review in Automatic Programming, Vol 5 (1969).

Alpert, D. and Bitzer, D. L., "Advances in Computer-based Education" in Science, 167 (1970), pp. 1582-1590.

Arnborg, Stefan, "A Note on the Assignment of Measurement Points for Frequency Counts in Structured Programs," BIT 14, (1974) pp. 273-278.

Barta, Ben Zion and Nievergelt, Jurg, "An Interactive System for Automatic Examination of Programming Skills (ISAEP)" to be published 1975.

Bernstein, A. J., "Analysis of Programs for Parallel Programming" IEEE Transactions on Computers, Vol. EC-15, No. 5 (Oct, 1966) pp. 757-763.

Biss, K., Chien, R. and Stahl, F., "R2-A Natural Language Question-answering System," AFIPS Conference Proceedings, Spring Joint Computer Conference, 1971, AFIPS Press, Montvale, N. J.

Bohm, Corrado and Jacopini, Giuseppe, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," Comm. of ACM, May, 1966, Vol. 9, No. 5.

Carbonell, Jaime R., "AI in CAI: An Artificial Intelligence Approach to Computer-Assisted Instruction," IEEE Transactions on Man-Machine Systems, Dec., 1970.

Charniak, Eugene, "Toward a Model of Children's Story Comprehension," AI TR-266, MIT Artificial Intelligence Laboratory, Cambridge, Mass., 1972.

Cooper, D. C., "Bohm and Jacopini's Reduction of Flowcharts," Comm. of ACM, Vol. 10, No. 8 (August, 1967).

Dahl, O. J., Dijkstra, E. W. and Hoare, C.A.R., Structured Programming, Academic Press, New York, 1972.

Danielson, Ronald L., Nievergelt, Jurg, "An Automatic Tutor for Introductory Programming Students." SIGCSE Bulletin, Feb. 1975, Vol. 7, No. 1, pp. 47-50.

Denning, Peter J., "Guest Editor's Overview," Computing Surveys Vol. 6, No. 4, Dec. 1974, pp. 209-211.

Derksen, J. A., Rulifson, J. F., Waldinger, R. J., "The QA4 Language Applied to Robot Planning," AFIPS Conference Proceedings, Vol. 41, Part 2, FJCC, 1972.

Dijkstra, Edgsger W., "GOTO Statement Considered Harmful," Comm. of ACM, Vol. 11, No. 3, March 1968.

Dijkstra, Edgsger W., "Notes on Structured Programming," T. H. Report 70 WSK-03, 2nd Edition, Technological University, Eindhoven, Netherlands, April 1970.

Dijkstra, Edgsger W., "The Humble Programmer," Comm. of ACM, Vol. 15, No. 10, Oct. 1972.

Ernst, G. W. and Newell A., GPS: A Case Study in Generality and Problem Solving, Academic Press, New York, 1969.

Floyd, Robert W., "Assigning Meaning to Programs," Proc. Symp. Appl. Math. 19, in J. T. Schwartz (ed.) Mathematical Aspects of Computer Science, American Mathematical Society, Providence, R. I., 1967.

Forsythe, G. E. and Wirth, N., "Automatic Grading of Programs," Comm. of ACM, 8 (1965), pp 275-278.

Foulk, Clinton R., "Smooth Programs," a talk at Computer Science Conference 1973, Columbus, Ohio, Feb. 21, 1973.

Gerhart, Susan L., "Methods for Teaching Program Verification," SIGCSE Bulletin, Vol. 7, No. 1 (1975), pp. 172-176.

Gries, David, "On Structured Programming--A Reply to Smoliar," Comm. of ACM, Vol. 17, No. 11, Nov. 1974, pp. 655-657.

Gries, David, "Research in Programming and Programming Languages," talk at ACM Computer Science Conference '75, Washington, D. C., Feb. 18, 1975.

Henschen, Lawrence J., "A Resolution Style Proof Procedure for Higher-Order Logic," DCL Report #452, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1971.

Hewitt, Carl, "Procedural Embedding of Knowledge in PLANNER," Proceedings of the Second International Joint Conference on Artificial Intelligence, London, 1971.

Hoare, C.A.R., "Proof of a Program: FIND," Comm. of ACM, 14, 1, Jan. 1971, pp. 39-45.

Hopcroft, John E., Ullman, Jeffrey D., Formal Languages and their Relation to Automa, Addison-Wesley Publishing Company, Reading, Mass., 1969.

Hyde, Daniel C., "Instructor's Manual for PASF," DCL Report UIUCDCS-R-75-744, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1975.

Igarashi, S., London, R. L. and Luckham, D. C., "Automatic Program Verification I: Logical Basis and Its Implementation," Computer Science Report 365, Stanford University, Stanford, Cal., May, 1973.

Kasai, Takumi, "Translatability of Flowcharts into WHILE Programs," Journal of Computer and System Science, 9 (1974), pp. 177-195.

Katz, S. M., and Manna, Z., "A Heuristic Approach to Program Verification." Third International Joint Conference on Artificial Intelligence, Stanford, Cal. (August, 1973), pp. 500-512.

Keller, Robert M., "A Solvable Program-Schema Equivalence Problem," Fifth Annual Princeton Conference on Information Science and Systems, March, 1971.

Knuth, Donald E., "Structured Programming with GOTO Statements," Computing Surveys, Vol. 6, No. 4, Dec. 1974, pp. 261-301.

Knuth, Donald E., The Art of Computer Programming, Vol. 1, 2nd Edition, Addison-Wesley Publishing Company, Reading, Mass., 1973.

Knuth, Donald E. and Floyd, Robert W., "Notes on Avoiding 'GOTO' Statements," Information Processing Letters, Vol. 1, No. 1 (Feb., 1971), pp. 23-31, 77.

Krause, K. W., Smith, R. W. and Goodwin, M. A., "Optimal Software Test Planning Through Automated Network Analysis," Record of 1973 IEEE Symposium on Computer Software Reliability, May, 1973, pp. 18-22.

Lee, John A. N., Computer Semantics: Studies of Algorithms, Processors and Languages, Van Nostrand Reinhold Co., N. Y., 1972.

London, Ralph L., "The Current State of Proving Programs Correct," National Proceedings of ACM, August, 1972, Vol. I, pp. 39-46.

Manna, Zohar, Mathematical Theory of Computation, McGraw-Hill Book Company, New York, 1974.

Manna, Z., Ness, S., and Vuillemin, J., "Inductive Methods for Proving Properties of Programs," Proc. of an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, Jan. 1972.

Mateti, Prabhaker, "A Sorting Program Verifier: A Tutoring System for Sorting Programs," Ph.D. Thesis Proposal, April 1974, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois.

McCarty, John, Turing Award Lecture, ACM National Conference, Chicago, Illinois, August, 1971.

McCormick, B. H., Ray, S. R., Smith, K. C. and Yamada, S., "Illiac III: A Processor of Visual Information," DCL Report #183, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, June 1965.

Michie, Donald (editor), Machine Intelligence I, American Elsevier, New York, 1967.

Mills, Harlan D., "The New Math of Computer Programming," Comm. of the ACM, Jan. 1975, Vol. 18, No. 1.

Minsky, Marvin, Editor, Semantic Information Processing, MIT Press, Cambridge, Mass., 1968.

Minsky, Marvin and Papert, Seymour, "Research at the Laboratory in Vision, Language, and Other Problems of Intelligence," AI Memo No. 252, MIT, Jan. 1972.

Montanari, Ugo G., "Separable Graphs, Planar Graphs and Web Grammars," Information and Control, 16, pp. 243-267 (1970) May, #3.

Nassi, I. and Shneiderman, Ben, "Flowchart Techniques for Structured Programming," SIGPLAN Notices, Vol. 8 #8, August 1973.

Naur, P., "Automatic Grading of Student's Algol Programming," BIT, 4 (1964), pp. 177-188.

Newell, Allen, Simon, Herbert A., Human Problem Solving, Prentice Hall, Englewood Cliffs, N. J., 1972.

Nievergelt, J., Reingold, E. M., and Wilcox, T. R., "The Automation of Introductory Computer Science Courses," in A. Gunther, et al. (Editors), International Computing Symposium 1973, North-Holland Publishing Co., 1974.

Nilsson, Nils J., Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, N. Y., 1971.

Osterweil, Leon J. and Fosdick, Lloyd D., "Data Flow Analysis as an Aid in Documentation, Assertion Generation, Validation and Error Detection," Report #CU-CS-055-74 Department of Computer Science, University of Colorado, Boulder, Colorado, 1974.

Pfaltz, John L., and Rosenfeld, Azriel, "Web Grammars," Proceedings International Joint Conference on Artificial Intelligence, Washington, D. C., May 7-9, 1969.

Ray, S. R., and Preparata, F., "An Approach to Artificial Non-symbolic Cognition," International Journal of Information Sciences, 4 1:65-86, 1972.

Robinson, J. A., "A Machine Oriented Logic Based on the Resolution Principle," Journal of the ACM, Vol. 12 (1965), pp. 23-41.

Ruth, Gregory R., "Intelligent Program Analysis," unpublished paper, MIT, Cambridge, Mass., Feb. 4, 1974.

Stifle, Jack, "The Plato IV Student Terminal," CERL X-15, University of Illinois at Urbana-Champaign, Urbana, Illinois, March 1970.

Sussman, Gerald Jay, "Teaching of Procedures," MIT AI Lab., Memo 270, Cambridge, Mass., Oct. 1972.

Sussman, G. J., Winograd, T., Charniak, E., "Micro-PLANNER Reference Manual," MIT AI Lab., Memo #203A, Dec. 1971.

Turing, Allan M., "On Computable Numbers with an Application to the Entscheidungsproblem," Proc. London Math. Soc., 2-42, pp. 230-265, 1936.

Waldinger, R. J. and Levitt, K. N., "Reasoning About Programs," Artificial Intelligence 5 (1974) pp. 235-316.

Wegbreit, B., "Heuristic Method for Mechanically Deriving Inductive Assertions," Third International Joint Conference on Artificial Intelligence, Stanford, Cal., pp. 524-536 (August, 1973)

Winograd, Terry, "Five Lectures on Artificial Intelligence," AI Lab., Memo AIM-246, Stanford University, Stanford, Cal., Sept. 1974.

Winograd, Terry, "Procedures as a Representation for Data in a Computer Program for Understanding Natural Language," Project Mac, MIT, MAC TR-84, MIT, Cambridge, Mass., 1971.

Wirth, N., "Program Development by Step-Wise Refinement," Comm. of ACM, Vol. 14, No. 4 (April, 1971), pp. 221-227.

Wirth, N., "The Remaining Trouble Spots in PASCAL," Department of Computer Science Colloquium, University of Illinois at Urbana-Champaign, April 7, 1975.

APPENDIX

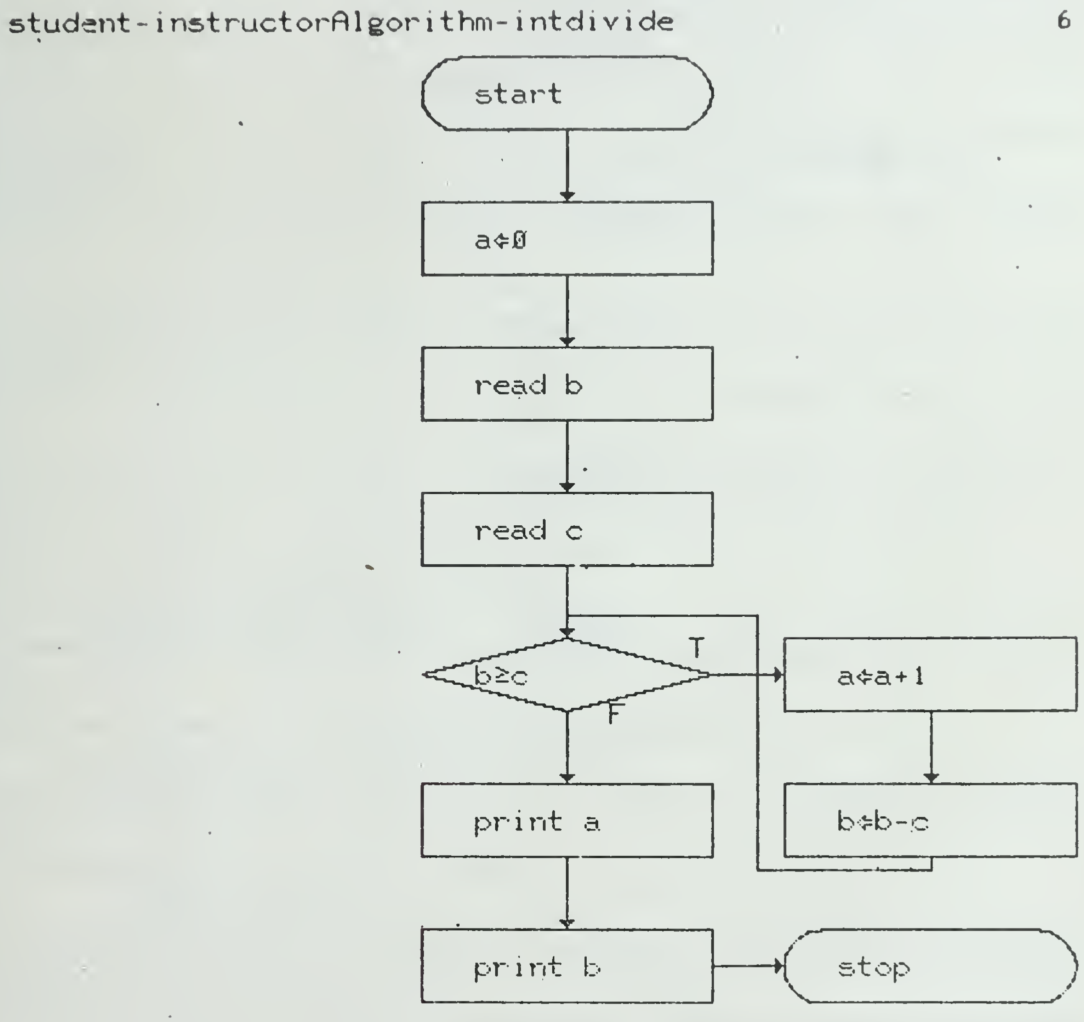
STEPS IN THE DEDUCTION SCHEME FOR THREE EXAMPLES

The Appendix includes three examples of student flowcharts to demonstrate the steps in the deduction scheme (cf. Section 3.7) of the Program to Analyze Smooth Flowcharts (PASF). The three examples are attempts by students to draw a smooth flowchart for the algorithm "intdivide" (integer divide by the method of successive subtraction).

The first flowchart (Fig. A-1) shows the smooth flowchart inputted by the instructor as the "correct" answer to "intdivide." The other three flowcharts (Figs. A-2, A-5, and A-11) are attempts by students "maryjane" and "dan" to do the algorithm "intdivide."

At the bottom of each flowchart is displayed the Regular Expression (RE) representation (cf. Section 3.6.4) of that flowchart. This is normally not displayed to students.

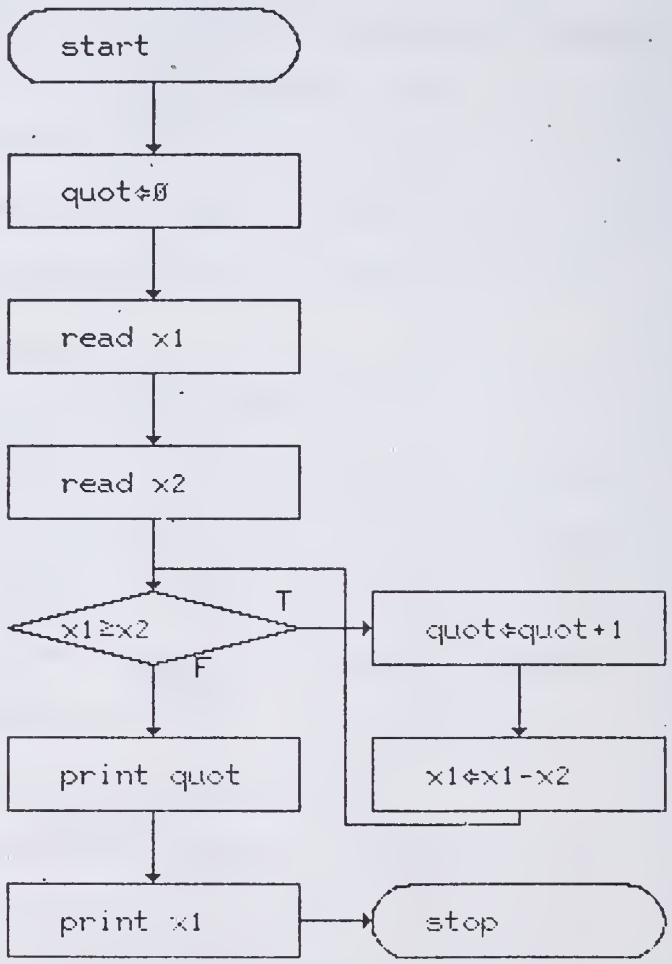
On the pages following each student's flowchart is displayed the steps in the deduction scheme (cf. Fig. 3-14) in which PASF tries to show that the instructor's flowchart and the student's flowchart are equivalent. The student's Standard Regular Expression (SRE) (cf. Section 3.7.1) is denoted by "$\underline{S}$." The instructor's SRE is denoted by "$\underline{I}$." To facilitate reading, the portion of each SRE which was changed from the line above is underlined. These displays of the steps in the deduction scheme were computer generated by the program specifically for this report and are <u>not</u> shown to the student.



RE=a←0 readb readc (a←a+1 b←b-c)*[b≥c]printa printb

Fig. A-1. Instructor's "correct" flowchart to algorithm "intdivide." At the bottom the Regular Expression (RE) representation of the flowchart is displayed.

```
student-maryjane  Algorithm-intdivide                          3
```



```
RE=quot←0 readx1 readx2 (quot←quot+1 x1←x1-x2 )*[x1≥x2 ]printquo
t printx1
```

Fig. A-2. Student "maryjane's" attempt at algorithm "intdivide." The flowchart is shown to be correct by the following two pages. At the bottom the Regular Expression (RE) representation of the flowchart is displayed.

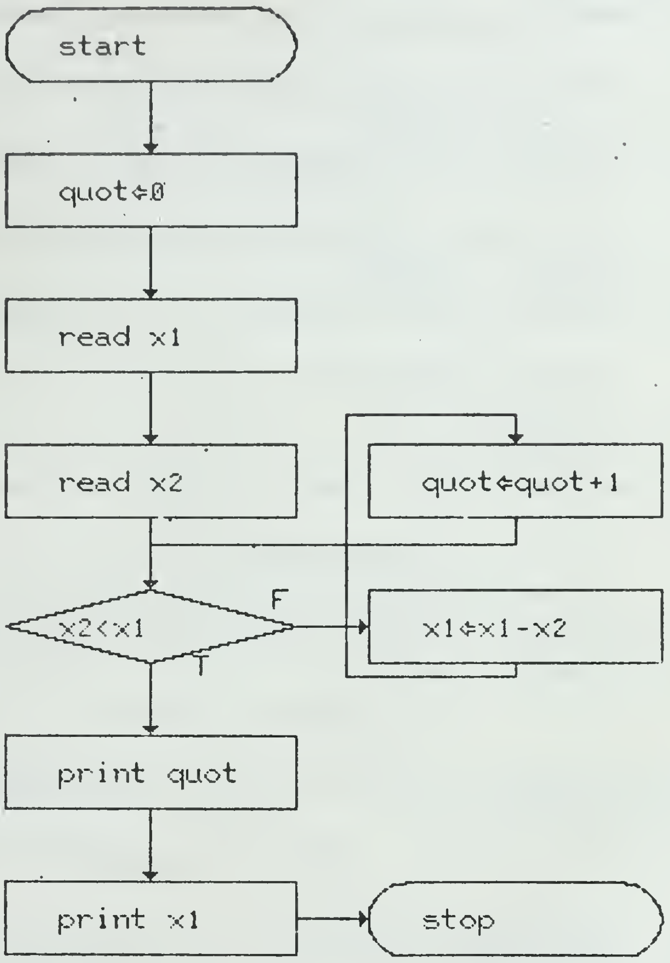
```
     Entering SELECTOR
S= intØ:quot; inp:x1; inp:x2; ( addquot1:quot; subx1x2:x1;) * [x1≥x
2] outquot:; outx1:;
I= intØ:a; inp:b; inp:c; ( adda1:a; subbc:b;) * [b≥c] outa:; outb:;

S= intØ:quot; inp:x1; inp:x2; ( P1 quot1:quot; subx1x2:x1;) * [x1≥x
2] outquot:; outx1:;
I= intØ:a; inp:b; inp:c; ( P1 a1:a; subbc:b;) * [b≥c] outa:; outb:;

S= intØ:quot; inp:x1; inp:x2; ( P1 quot1:quot; P2 x1x2:x1;) * [x1≥x
2] outquot:; outx1:;
I= intØ:a; inp:b; inp:c; ( P1 a1:a; P2 bc:b;) * [b≥c] outa:; outb:;

     Entering REDUCOR
S= intØ:quot; inp:x1; inp:x2; ( P1 quot1:quot; P2 x1x2:x1;) * [x1≥x
2] outquot:; outx1:;
I= intØ:a; inp:b; inp:c; ( P1 a1:a; P2 bc:b;) * [b≥c] outa:; outb:;

S= intØ:quot; inp:x1; inp:x2; ( P3 quot1x1x2:quotx1;) * [x1≥x2] out
quot:; outx1:;
I= intØ:a; inp:b; inp:c; ( P3 a1bc:ab;) * [b≥c] outa:; outb:;

     Entering SELECTOR
S= intØ:quot; inp:x1; inp:x2; P4 x1x2quot1x1x2:;quotx1 outquot:;
 outx1:;
I= intØ:a; inp:b; inp:c; P4 bca1bc:;ab outa:; outb:;

S= intØ:quot; P5 :x1; inp:x2; P4 x1x2quot1x1x2:;quotx1 outquot:;
 outx1:;
I= intØ:a; P5 :b; inp:c; P4 bca1bc:;ab outa:; outb:;

S= intØ:quot; P5 :x1; P6 :x2; P4 x1x2quot1x1x2:;quotx1 outquot:;
 outx1:;
I= intØ:a; P5 :b; P6 :c; P4 bca1bc:;ab outa:; outb:;

S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x1x2quot1x1x2:;quotx1 outquot:;
 outx1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab outa:; outb:;
```

Fig. A-3. Steps in the deduction scheme for the flowchart of Fig. A-2.

```
S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x1x2quot1x1x2:;quotx1 P8 quot:;
 outx1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P8 a:; outb:;

S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x1x2quot1x1x2:;quotx1 P8 quot:;
 P9 x1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P8 a:; P9 b:;

     Entering REDUCOR
S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x1x2quot1x1x2:;quotx1 P8 quot:;
 P9 x1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P8 a:; P9 b:;

S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x1x2quot1x1x2:;quotx1 P1Ø quotx
1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P1Ø ab:;

S= P11 Ø:quotx1; P6 :x2; P4 x1x2quot1x1x2:;quotx1 P1Ø quotx1:;

I= P11 Ø:ab; P6 :c; P4 bca1bc:;ab P1Ø ab:;

S= P12 Ø:quotx1x2; P4 x1x2quot1x1x2:;quotx1 P1Ø quotx1:;

I= P12 Ø:abc; P4 bca1bc:;ab P1Ø ab:;

S= P13 Ø1:quotx1x2; P1Ø quotx1:;

I= P13 Ø1:abc; P1Ø ab:;

S= P14 Ø1:quotx1x2;

I= P14 Ø1:abc;

Your flowchart is correct for algorithm intdivide.
Press -NEXT- to do another algorithm.
```

Fig. A-4. Continuation in the steps in the deduction scheme for flowchart of Fig. A-2.

student-dan   Algorithm-intdivide   7



```
RE=quot⇐0 readx1 readx2 (x1⇐x1-x2 quot⇐quot+1 )* [x2≥x1 ]printquo
t printx1
```

Fig. A-5. Student "dan's" attempt at algorithm "intdivide." The flowchart is shown to be not correct by the following five pages (diamond should have x1 < x2). At the bottom the Regular Expression (RE) representation of the flowchart is shown.

```
     Entering SELECTOR
S= int0:quot; inp:x1; inp:x2;( subx1x2:x1; addquot1:quot;)*[x2≥x
1] outquot:; outx1:;
I= int0:a; inp:b; inp:c;( adda1:a; subbc:b;)*[b≥c] outa:; outb:;

S= int0:quot; inp:x1; inp:x2;( subx1x2:x1; P1 quot1:quot;)*[x2≥x
1] outquot:; outx1:;
I= int0:a; inp:b; inp:c;( P1 a1:a; subbc:b;)*[b≥c] outa:; outb:;

S= int0:quot; inp:x1; inp:x2;( P2 x1x2:x1; P1 quot1:quot;)*[x2≥x
1] outquot:; outx1:;
I= int0:a; inp:b; inp:c;( P1 a1:a; P2 bc:b;)*[b≥c] outa:; outb:;

     Entering REDUCOR
S= int0:quot; inp:x1; inp:x2;( P2 x1x2:x1; P1 quot1:quot;)*[x2≥x
1] outquot:; outx1:;
I= int0:a; inp:b; inp:c;( P1 a1:a; P2 bc:b;)*[b≥c] outa:; outb:;

     Entering TRANSLATOR B
     Entering REDUCOR after interchange in TRANSLATOR B
S= int0:quot; inp:x1; inp:x2;( P1 quot1:quot; P2 x1x2:x1;)*[x2≥x
1] outquot:; outx1:;
I= int0:a; inp:b; inp:c;( P1 a1:a; P2 bc:b;)*[b≥c] outa:; outb:;

S= int0:quot; inp:x1; inp:x2;( P3 quot1x1x2:quotx1;)*[x2≥x1] out
quot:; outx1:;
I= int0:a; inp:b; inp:c;( P3 a1bc:ab;)*[b≥c] outa:; outb:;

     Entering SELECTOR
S= int0:quot; inp:x1; inp:x2; P4 x2x1quot1x1x2:;quotx1 outquot:;
 outx1:;

I= int0:a; inp:b; inp:c; P4 bca1bc:;ab outa:; outb:;

S= int0:quot; P5 :x1; inp:x2; P4 x2x1quot1x1x2:;quotx1 outquot:;
 outx1:;
I= int0:a; P5 :b; inp:c; P4 bca1bc:;ab outa:; outb:;
```

Fig. A-6. Steps in the deduction scheme for the flowchart of Fig. A-5.

```
S= intØ:quot; P5 :x1; P6 :x2; P4 x2x1quot1x1x2:;quotx1 outquot:;
 outx1:;
I= intØ:a; P5 :b; P6 :c; P4 bca1bc:;ab outa:; outb:;

S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x2x1quot1x1x2:;quotx1 outquot:;
 outx1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab outa:; outb:;

S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x2x1quot1x1x2:;quotx1 P8 quot:;
 outx1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P8 a:; outb:;

S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x2x1quot1x1x2:;quotx1 P8 quot:;
 P9 x1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P8 a:; P9 b:;

     Entering REDUCOR
S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x2x1quot1x1x2:;quotx1 P8 quot:;
 P9 x1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P8 a:; P9 b:;

S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x2x1quot1x1x2:;quotx1 P1Ø quotx
1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P1Ø ab:;

S= P11 Ø:quotx1; P6 :x2; P4 x2x1quot1x1x2:;quotx1 P1Ø quotx1:;

I= P11 Ø:ab; P6 :c; P4 bca1bc:;ab P1Ø ab:;

S= P12 Ø:quotx1x2; P4 x2x1quot1x1x2:;quotx1 P1Ø quotx1:;

I= P12 Ø:abc; P4 bca1bc:;ab P1Ø ab:;

     Backtrack has taken place; stack popped
S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x2x1quot1x1x2:;quotx1 outquot:;
 outx1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab outa:; outb:;
```

Fig. A-7. Continuation in the steps in the deduction scheme for flowchart in Fig. A-5.

```
     Entering SELECTOR
S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x2x1quot1x1x2:;quotx1 outx1:; o
utquot:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab outa:; outb:;

S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x2x1quot1x1x2:;quotx1 P8 x1:; o
utquot:;
I= P7 Ø:a; P5 ;b; P6 :c; P4 bca1bc:;ab P8 a:; outb:;

S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x2x1quot1x1x2:;quotx1 P8 x1:; P
9 quot:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P8 a:; P9 b:;

     Entering REDUCOR
S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x2x1quot1x1x2:;quotx1 P8 x1:; P
9 quot:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P8 a:; P9 b:;

S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x2x1quot1x1x2:;quotx1 P1Ø x1quo
t:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P1Ø ab:;

S= P11 Ø:quotx1; P6 :x2; P4 x2x1quot1x1x2:;quotx1 P1Ø x1quot:;

I= P11 Ø:ab; P6 :c; P4 bca1bc:;ab P1Ø ab:;

S= P12 Ø:quotx1x2; P4 x2x1quot1x1x2:;quotx1 P1Ø x1quot:;

I= P12 Ø:abc; P4 bca1bc:;ab P1Ø ab:;

     Backtrack has taken place; stack popped
S= intØ:quot; inp:x1; inp:x2; P4 x2x1quot1x1x2:;quotx1 outquot:;
 outx1:;
I= intØ:a; inp:b; inp:c; P4 bca1bc:;ab outa:; outb:;

     Entering SELECTOR
S= intØ:quot; inp:x2; inp:x1; P4 x2x1quot1x1x2:;quotx1 outquot:;
 outx1:;
I= intØ:a; inp:b; inp:c; P4 bca1bc:;ab outa:; outb:;
```

Fig. A-8. Continuation in the steps in the deduction scheme for flowchart in Fig. A-5.

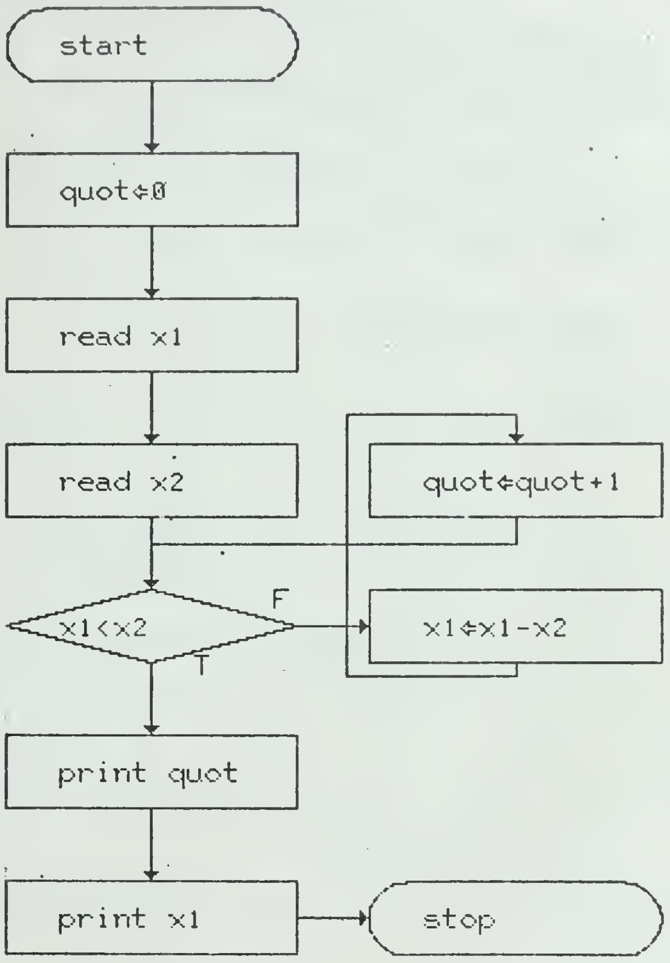
```
S= intØ:quot; P5 :x2; inp:x1; P4 x2x1quot1x1x2:;quotx1 outquot:;
 outx1:;
I= intØ:a; P5 :b; inp:c; P4 bca1bc:;ab outa:; outb:;

S= intØ:quot; P5 :x2; P6 :x1; P4 x2x1quot1x1x2:;quotx1 outquot:;
 outx1:;
I= intØ:a; P5 :b; P6 :c; P4 bca1bc:;ab outa:; outb:;

S= P7 Ø:quot; P5 :x2; P6 :x1; P4 x2x1quot1x1x2:;quotx1 outquot:;
outx1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab outa:; outb:;

S= P7 Ø:quot; P5 :x2; P6 :x1; P4 x2x1quot1x1x2:;quotx1 P8 quot:;
 outx1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P8 a:; outb:;

S= P7 Ø:quot; P5 :x2; P6 :x1; P4 x2x1quot1x1x2:;quotx1 P8 quot:;
 P9 x1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P8 a:; P9 b:;

     Entering REDUCOR
S= P7 Ø:quot; P5 :x2; P6 :x1; P4 x2x1quot1x1x2:;quotx1 P8 quot:;
 P9 x1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P8 a:; P9 b:;

S= P7 Ø:quot; P5 :x2; P6 :x1; P4 x2x1quot1x1x2:;quotx1 P1Ø quotx
1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P1Ø ab:;

S= P11 Ø:quotx2; P6 :x1; P4 x2x1quot1x1x2:;quotx1 P1Ø quotx1:;

I= P11 Ø:ab; P6 :c; P4 bca1bc:;ab P1Ø ab:;

S= P12 Ø:quotx2x1; P4 x2x1quot1x1x2:;quotx1 P1Ø quotx1:;

I= P12 Ø:abc; P4 bca1bc:;ab P1Ø ab:;

     Backtrack has taken place; stack pooped
S= P7 Ø:quot; P5 :x2; P6 :x1; P4 x2x1quot1x1x2:;quotx1 outquot:;
 outx1:;
```

Fig. A-9. Continuation in the steps in the deduction scheme for flowchart in Fig. A-5.

```
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab outa:; outb:;

     Entering SELECTOR
S= P7 Ø:quot; P5 :x2; P6 :x1; P4 x2x1quot1x1x2:;quotx1 outx1:; o
utquot:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab outa:; outb:;

S= P7 Ø:quot; P5 :x2; P6 :x1; P4 x2x1quot1x1x2:;quotx1 P8 x1:; o
utquot:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P8 a:; outb:;

S= P7 Ø:quot; P5 :x2; P6 :x1; P4 x2x1quot1x1x2:;quotx1 P8 x1:; P
9 quot:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P8 a:; P9 b:;

     Entering REDUCOR
S= P7 Ø:quot; P5 :x2; P6 :x1; P4 x2x1quot1x1x2:;quotx1 P8 x1:; P
9 quot:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P8 a:; P9 b:;

S= P7 Ø:quot; P5 :x2; P6 :x1; P4 x2x1quot1x1x2:;quotx1 P1Ø x1quo
t:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P1Ø ab:;

S= P11 Ø:quotx2; P6 :x1; P4 x2x1quot1x1x2:;quotx1 P1Ø x1quot:;

I= P11 Ø:ab; P6 :c; P4 bca1bc:;ab P1Ø ab:;

S= P12 Ø:quotx2x1; P4 x2x1quot1x1x2:;quotx1 P1Ø x1quot:;

I= P12 Ø:abc; P4 bca1bc:;ab P1Ø ab:;

Your flowchart is NOT correct for algorithm intdivide.
Press -NEXT- for a list of POSSIBLE errors
```

Fig. A-10. Continuation in the steps in the deduction scheme for flowchart in Fig. A-5.

student-dan   Algorithm-intdivide   8



RE=quot←Ø readx1 readx2 (x1←x1-x2 quot←quot+1)*[x1≥x2]printquo
t printx1

Fig. A-11. Student "dan's" attempt at algorithm "intdivide." The flowchart is shown to be correct by the following two pages. At the bottom the Regular Expression (RE) representation of the flowchart is displayed.

```
     Entering SELECTOR
S= int0:quot; inp:x1; inp:x2;( subx1x2:x1; addquot1:quot;)*[x1≥x
2] outquot:; outx1:;
I= int0:a; inp:b; inp:c;( adda1:a; subbc:b;)*[b≥c] outa:; outb:;

S= int0:quot; inp:x1; inp:x2;( subx1x2:x1; P1 quot1:quot;)*[x1≥x
2] outquot:; outx1:;
I= int0:a; inp:b; inp:c;( P1 a1:a; subbc:b;)*[b≥c] outa:; outb:;

S= int0:quot; inp:x1; inp:x2;( P2 x1x2:x1; P1 quot1:quot;)*[x1≥x
2] outquot:; outx1:;
I= int0:a; inp:b; inp:c;( P1 a1:a; P2 bc:b;)*[b≥c] outa:; outb:;

     Entering REDUCOR
S= int0:quot; inp:x1; inp:x2;( P2 x1x2:x1; P1 quot1:quot;)*[x1≥x
2] outquot:; outx1:;
I= int0:a; inp:b; inp:c;( P1 a1:a; P2 bc:b;)*[b≥c] outa:; outb:;

     Entering TRANSLATOR B
     Entering REDUCOR after interchange in TRANSLATOR B
S= int0:quot; inp:x1; inp:x2;( P1 quot1:quot; P2 x1x2:x1;)*[x1≥x
2] outquot:; outx1:;
I= int0:a; inp:b; inp:c;( P1 a1:a; P2 bc:b;)*[b≥c] outa:; outb:;

S= int0:quot; inp:x1; inp:x2;( P3 quot1x1x2:quotx1;)*[x1≥x2] out
quot:; outx1:;
I= int0:a; inp:b; inp:c;( P3 a1bc:ab;)*[b≥c] outa:; outb:;

     Entering SELECTOR
S= int0:quot; inp:x1; inp:x2; P4 x1x2quot1x1x2::quotx1 outquot:;
 outx1:;

I= int0:a; inp:b; inp:c; P4 bca1bc:;ab outa:; outb:;

S= int0:quot; P5 :x1; inp:x2; P4 x1x2quot1x1x2:;quotx1 outquot:;
 outx1:;
I= int0:a; P5 :b; inp:c; P4 bca1bc:;ab outa:; outb:;
```

Fig. A-12. Steps in the deduction scheme for the flowchart in Fig. A-11.

```
S= intØ:quot; P5 :x1; P6 :x2; P4 x1x2quot1x1x2:;quotx1 outquot:;
 outx1:;
I= intØ:a; P5 :b; P6 :c; P4 bca1bc:;ab outa:; outb:;

S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x1x2quot1x1x2:;quotx1 outquot:;
 outx1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab outa:; outb:;

S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x1x2quot1x1x2:;quotx1 P8 quot:;
 outx1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P8 a:; outb:;

S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x1x2quot1x1x2:;quotx1 P8 quot:;
 P9 x1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P8 a:; P9 b:;

     Entering REDUCOR
S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x1x2quot1x1x2:;quotx1 P8 quot:;
 P9 x1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P8 a:; P9 b:;

S= P7 Ø:quot; P5 :x1; P6 :x2; P4 x1x2quot1x1x2:;quotx1 P1Ø quotx
1:;
I= P7 Ø:a; P5 :b; P6 :c; P4 bca1bc:;ab P1Ø ab:;

S= P11 Ø:quotx1; P6 :x2; P4 x1x2quot1x1x2:;quotx1 P1Ø quotx1:;

I= P11 Ø:ab; P6 :c; P4 bca1bc:;ab P1Ø ab:;

S= P12 Ø:quotx1x2; P4 x1x2quot1x1x2:;quotx1 P1Ø quotx1:;

I= P12 Ø:abc; P4 bca1bc:;ab P1Ø ab:;

S= P13 Ø1:quotx1x2; P1Ø quotx1:;

I= P13 Ø1:abc; P1Ø ab:;

S= P14 Ø1:quotx1x2;

I= P14 Ø1:abc;
 Your flowchart is correct for algorithm intdivide.
 Press -NEXT- to do another algorithm.
```

Fig. A-13. Continuation of the steps in the deduction scheme for flowchart in Fig. A-11.

VITA

Daniel Clair Hyde was born in LeRoy, New York on March 18, 1946. He received his Bachelor of Science degree cum laude in Electrical Engineering from Northeastern University, Boston, Massachusetts in 1969. As a participant in the cooperative work program at Northeastern University, he was employed at Taylor Instrument Companies, Rochester, New York, and the Office of Academic Research, Northeastern University.

He joined the University of Illinois at Urbana-Champaign as a graduate teaching assistant in the Department of Computer Science in 1969. While a teaching assistant, he taught courses in introductory programming, introduction to theory of digital machines, design of digital switching circuits, and computer aided instruction. In 1971, he was awarded an NDEA Title IV Fellowship for two years. He conducted research in the areas of computer-based education, artificial intelligence, and software engineering under the guidance of Professor Sylvian R. Ray. From 1973 to 1975 he was employed by the Computer-based Education Research Laboratory to conduct research on the PLATO IV Project in the area of military training. He is the coauthor with Professor Bruce L. Hicks of a paper entitled, "Teaching about CAI," which was published in Journal of Teacher Education, Summer of 1973. He has been honored by membership in Eta Kappa Nu and Tau Beta Pi and by associate membership in Sigma Xi honor societies. He is a member of Institute of Electrical and Electronics Engineers and Association for Computing Machinery.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-75-743	2.	3. Recipient's Accession No.
4. Title and Subtitle ANALYZING SMOOTH FLOWCHARTS: TEACHING STRUCTURED PROGRAMMING IN A COMPUTER-BASED EDUCATION ENVIRONMENT			5. Report Date June, 1975
			6.
7. Author(s) Daniel Clair Hyde			8. Performing Organization Rept. No.
9. Performing Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61820			10. Project/Task/Work Unit No.
			11. Contract/Grant No.
12. Sponsoring Organization Name and Address			13. Type of Report & Period Covered PhD Thesis
			14.

15. Supplementary Notes

16. Abstracts

A method to machine grade student-generated structured programs is described. This method (Semantic Formulation Method)(SFM) is compared and contrasted with two other methods of grading programs: the test data (testing the program by inputting data) and the inductive assertion methods. It is shown that the SFM is ideally suited for grading programs in a computer-based education environment. A computer program implementing the SFM on the PLATO IV Computer-based Education System is described.
The class of structured programs utilized in the PLATO IV implementation version is smooth flowcharts. Smooth flowcharts are a subset of flowcharts which only allow the the DOWHILE and IFTHENELSE in their control structure.
The SFM of grading structured programs is based on the artificial intelligence paradigm of "description, representation and deduction." The special purpose deduction scheme is tailored to prove in an efficient manner that two structured programs (the student's and the instructor's) are equivalent.

17. Key Words and Document Analysis. 17a. Descriptors

algorithm development
artificial intelligence
computer assisted instruction
computer-based education
correctness
deduction scheme
D-flowcharts
equivalence of programs
flowcharts
grading programs
machine grading
PLATO
software engineering
stepwise refinement
structured flowcharts
structured programming
teaching programming

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 220
	20. Security Class (This Page UNCLASSIFIED	22. Price